

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

Python

程序设计教程

3.5

涵盖Python 3.0到3.5实用特性
详细介绍函数、模块、类与组件管理
深入探讨常用模块的应用与实践
包含装饰器、meta类的实践等进阶主题

林信良 著



清华大学出版社



内 容 简 介

本书是作者在Python开发应用中的经验总结，旨在帮助读者快速掌握Python编程。全书共分10章，从Python的入门知识开始，逐步深入到高级应用。第1章介绍Python的起源、安装和运行环境。第2章讲解Python的基本数据类型和运算符。第3章介绍Python的列表、字典等容器类型。第4章讲解Python的字符串操作。第5章介绍Python的元组、集合和文件操作。第6章讲解Python的函数和模块。第7章介绍Python的异常处理。第8章讲解Python的面向对象编程。第9章介绍Python的数据库操作。第10章介绍Python的Web应用开发。本书适合Python初学者阅读，也可作为相关课程的教材。

Python

程序设计教程

林信良 著

清华大学出版社
地址：北京清华大学学研大厦A座
邮编：100084
电话：(010) 62770175
网址：http://www.tup.com.cn, http://www.wqbook.com

清华大学出版社
地址：北京清华大学学研大厦A座
邮编：100084
电话：(010) 62770175
网址：http://www.tup.com.cn, http://www.wqbook.com

清华大学出版社
地址：北京清华大学学研大厦A座
邮编：100084
电话：(010) 62770175
网址：http://www.tup.com.cn, http://www.wqbook.com

清华大学出版社
北京

产品编号：0132376-01

内 容 简 介

本书是作者在 Python 教学中学生在课程上遇到的概念、实战、应用等问题的经验总结。

本书基于 Python 3.5 编写,介绍了 Python 3.0 到 3.5 的实用特性。本书用简短精巧的范例程序贯穿全书,以学习笔记的写作方式进行编写,让读者在 Python 语言的交互环境中直接动手实战和体验,通过“实战”来掌握 Python 语言的核心知识和实战用法,并且特意标注了常用范例和重点范例,让读者可以根据自己的时间安排进行取舍。

本书既适合初学者学习,又能帮助有一定基础的程序员提升技能,还可作为相关培训的教材。

本书为碁峰资讯股份有限公司授权出版发行的中文简体字版本。

北京市版权局著作权合同登记号 图字:01-2016-8572

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

Python 程序设计教程/林信良著.—北京:清华大学出版社,2017

ISBN 978-7-302-45786-2

I. ①P... II. ①林... III. ①软件工具—程序设计—教材 IV. ①TP311.56

中国版本图书馆 CIP 数据核字(2016)第 290544 号

责任编辑:夏毓彦

封面设计:王翔

责任校对:闫秀华

责任印制:何芊

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦 A 座 邮 编:100084

社总机:010-62770175 邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

印装者:三河市春园印刷有限公司

经 销:全国新华书店

开 本:190mm×260mm

印 张:22.75

字 数:580 千字

版 次:2017 年 1 月第 1 版

印 次:2017 年 1 月第 1 次印刷

印 数:1~3000

定 价:59.00 元

产品编号:072376-01

序

你会不会做实验呢？

当然，对于程序设计来说，懂得做实验是件很重要的事。看到一个函数特性，编写个范例程序来实验看看；不懂某条语句及其语法的概念，编写个程序片段来实验看看，虽然这不是在做化学实验或物理实验，不过有时也会想实验看看，程序会不会像化学实验那样“炸”了……

关掉计算机之后，你会不会做实验呢？

愿意从事程序设计的人，按理来说，应该也是乐于做实验的人，那么现在就做个实验吧！关掉计算机、离开桌子，想想除了用计算机之外，还能在生活上做些什么实验？或者尝试看看其他事物，看看会有什么样的结果。

有没有对自己的人生做过实验呢？

这和计算机上做实验不同，对人生做实验需要耐心，没有人能保证何时能有结果，有时人生中看似毫不相关，甚至是失败的几个实验，却在某个时间点获得莫名其妙的成果。

有没有特意对未来的人生进行实验呢？

你回想起过去曾经有过的几次实验，也许算不上实验，只是在随波逐流的过程中，多少尝试过做些努力，若在无意识下曾经对人生做过的实验促成了现在的你，那么现在下意识地对自己做些实验，未来的自己会是什么样子呢？

程序设计很强调 Get your hands dirty（要勤写代码），别忘了，人生也需要 Get your hands dirty（亲力亲为）！

编者

2016年9月

改编说明

作为具有“胶水”美誉的程序设计语言——Python，它广受欢迎的原因众多，现只列其二：

一、清晰简洁的动态类型语言。让初学者可以把精力集中在编程的对象以及编程的逻辑思维上，不用去担心程序设计语言的语法、数据类型、数据结构等各种烦人的内容。对于没有面向对象程序设计基础的人而言，直接学 C++ 或 Java 可能困难较大，从 Python 语言入手不失为一种更好的选择。

二、博取众长，兼顾 C++ 的高效率和 Java 网页设计的灵活性。即便对于已经掌握了 C++ 或 Java 等语言的人员，在认识了 Python 的超级“粘合剂”的作用之后，都会对它的能力赞叹不已。

因此，Python 不仅在专业人员中流行，越来越多程序设计初学者也开始把 Python 作为自己的第一门程序设计语言来学习。在高等院校采用 Python 作为学生的第一门程序设计语言也成为了最新的趋势。

本书的结构与叙述风格既像传统的程序设计语言教材，又不像。这是因为本书没有通篇“枯燥乏味”的讲解程序设计语言的语法，而且以“学会说一种流利的语言，而不是成为这种语言的语法专家”的原则来展开。学好一门语言的最好方式不是背语法，而是不断地练习怎么“说”——本书用简短精巧的示范和范例程序贯穿全书，让读者在 Python 语言的交互环境中直接动手实践和体验，通过“实战”来掌握 Python 语言的核心知识和实战用法。

为了配合本书作为学习和培训 Python 语言的教材（包括自学），除了各个章节给读者提供实践范例程序和教学范例程序，我们还为每章课后练习提供了解答的参考程序，且提供源代码。

最后加一点，如果读者按照本书介绍的步骤安装和设置 Python 的命令行或者集成开发的运行环境，那么默认的工作目录是 C:\workspace。如果本书提供的范例程序出现不能运行的情况，请注意设置好 Python 的运行环境和工作目录。

最后祝大家学习顺利，早日对 Python 语言融合贯通，展示自己的聪明才智！

资深架构师 赵军

2016 年 10 月

导 读

这份导读可以让你更了解如何使用本书。

程序范例

本书的范例程序下载地址为：<http://pan.baidu.com/s/1kVvCdEJ>（注意区分数字和英文字母大小写）。如果下载有问题，请发送电子邮件至 booksaga@126.com，邮件主题设置为“求 Python 程序设计教程下载资源”。

本书许多范例都使用完整的程序实现来展现，当你看到以下程序代码范例：

basicio upper.py


```
import sys

src_path = sys.argv[1]
dest_path = sys.argv[2]

with open(src_path) as src, open(dest_path, 'w') as dest:
    content = src.read()
    dest.write(content.upper())
```

① 分别以'r'与'w'模式打开
② 使用 read()读取数据
③ 使用 write()写入数据

范例开始的左边名称为 **basicio**，表示可以在范例文件夹的 **samples** 文件夹内，按各个章节的文件夹找到对应的 **basicio** 项目，右边名称为 **upper.py**，表示可以在项目找到 **upper.py** 文件。如果程序代码中出现标号与提示文字，表示后续的内容中会有对应于标号和提示的更详细的说明。

建议读者每个项目范例都亲自动手编写，但如果教学时间或实践时间不足的问题，本书也提供了建议进行的练习，如果在范例开始前发现有个  图标，例如：

game1 rpg.py

```
class Role:
    def __init__(self, name, level, blood):
        self.name = name # 角色名称
        self.level = level # 角色等级
        self.blood = blood # 角色血量

    def __str__(self):
        return "('{name}', {level}, {blood})".format(**vars(self))

    def __repr__(self):
        return self.__str__()

class SwordsMan(Role):
    def fight(self):
        print('挥剑攻击')

class Magician(Role):
    def fight(self):
```

① 定义类 Role
② 继承父类 Role
③ 继承父类 Role



```
print('魔法攻击')

def cure(self):
    print('魔法治疗')
```

表示建议动手实现这个范例进行上机实践，而且在范例文件 `labs` 文件夹中会有练习项目的基础，可以打开项目并完成项目中遗漏或必须补齐的程序代码或设置。

如果文中采用以下的程序代码排版，表示它是个代码段，主要展现程序编写时需要特别注意的片段。

```
class Account:
    ...
    def deposit(self, amount):
        if amount <= 0:
            print('存款金额不得为负')
        else:
            self.balance += amount

    def withdraw(self, amount):
        if amount > self.balance:
            print('余额不足')
        else:
            self.balance -= amount
```

对话框

在本书中会出现以下对话框：

提示 >>>

针对课程中所提到的概念，提供一些额外的资源或思考方向，暂时忽略这些提示对课程并没有影响，如果有时间，对这些提示多加阅读、思考和讨论将对本书的学习有帮助。

注意 >>>

对课程中所提到的概念，以对话框的方式特别呈现出必须注意的使用方式、陷阱或避开问题的方法，看到这个对话框时请集中精神仔细阅读。

附录

附录中的内容不是本书编写的主要目的，纯粹是让有兴趣的读者知道有这类主题的存在，因而附录的内容简明扼要。附录 A 简要介绍使用 Python 3.3 之后内建的 `venv` 模块来建立虚拟环境，附录 B 简要介绍如何使用 Django 编写简单的 Web 应用程序。

联系作者

若有勘误和反馈等和本书有关的问题，可通过网站与作者联系：

<http://openhome.cc>

目 录

第 1 章 Python 起步走	1
1.1 认识 Python.....	2
1.1.1 Python 3 的诞生	2
1.1.2 从 Python 3.0 到 3.5	3
1.1.3 初识 Python 的社区资源	5
1.2 建立 Python 环境	6
1.2.1 Python 的实现	6
1.2.2 下载与安装 Python 3.5	8
1.2.3 认识安装的内容	10
1.3 重点复习	12
第 2 章 从 REPL 到 IDE	14
2.1 从 'Hello World' 开始	15
2.1.1 使用 REPL	15
2.1.2 编写 Python 源码	18
2.1.3 哈啰! 世界!	20
2.2 初识模块与软件包	23
2.2.1 模块简介	23
2.2.2 设置 PYTHONPATH	25
2.2.3 使用软件包管理模块	27
2.2.4 使用 import as 与 from import	28
2.3 使用 IDE	29
2.3.1 下载、安装 PyCharm	29
2.3.2 IDE 项目管理基础	31
2.4 重点复习	35
第 3 章 类型与运算符	36
3.1 内建类型	37
3.1.1 数值类型	37
3.1.2 字符串类型	39
3.1.3 群集类型	45
3.2 变量与运算符	50

3.2.1	变量	50
3.2.2	加减乘除运算	52
3.2.3	比较与赋值运算	56
3.2.4	逻辑运算	57
3.2.5	位运算	58
3.2.6	索引切片运算	60
3.3	重点复习	62
	课后练习	64
第 4 章	流程语句与函数	65
4.1	流程语句	66
4.1.1	if 分支判断	66
4.1.2	while 循环	68
4.1.3	for in 迭代	70
4.1.4	pass、break、continue	72
4.1.5	for Comprehension	72
4.2	定义函数	74
4.2.1	使用 def 定义函数	75
4.2.2	参数与自变量	76
4.2.3	一级函数的运用	79
4.2.4	lambda 表达式	83
4.2.5	初探变量作用域	84
4.2.6	yield 与 yield from	87
4.3	重点复习	90
	课后练习	91
第 5 章	从模块到类	93
5.1	模块管理	94
5.1.1	用模块建立抽象层	94
5.1.2	管理模块名称	96
5.1.3	设置 PTH 文件	99
5.2	初识面向对象	101
5.2.1	定义类	101
5.2.2	定义方法	102
5.2.3	定义内部属性	105
5.2.4	定义外部属性	106
5.3	类语法的细节	108

5.3.1	绑定与未绑定方法	108
5.3.2	静态方法与类方法	110
5.3.3	属性命名空间	111
5.3.4	定义运算符	114
5.3.5	__new__()、__init__()与__del__()	116
5.4	重点复习	118
	课后练习	120
第 6 章	类的继承	121
6.1	何谓继承	122
6.1.1	继承共同行为	122
6.1.2	鸭子类型	124
6.1.3	重新定义方法	125
6.1.4	定义抽象方法	126
6.2	继承语法的细节	128
6.2.1	初识 object 与 super()	128
6.2.2	Rich comparison 方法	130
6.2.3	使用 enum 枚举	132
6.2.4	多重继承	134
6.2.5	创建 ABC (抽象基类)	136
6.2.6	探讨 super()	138
6.3	文档与软件包资源	141
6.3.1	DocStrings	142
6.3.2	查询官方文档	145
6.3.3	PyPI 与 pip	146
6.4	重点复习	147
	课后练习	148
第 7 章	例外处理	149
7.1	语法与继承结构	150
7.1.1	使用 try、except	150
7.1.2	例外继承结构	153
7.1.3	引发 (raise) 例外	155
7.1.4	Python 例外风格	159
7.1.5	认识堆栈追踪	160
7.1.6	提出警告信息	163
7.2	例外与资源管理	165

12.3 性能	288
12.3.1 timeit 模块	288
12.3.2 使用 cProfile (profile)	290
12.4 重点复习	292
课后练习	293
第 13 章 并发与并行	294
13.1 并发	295
13.1.1 线程简介	295
13.1.2 线程的启动与停止	297
13.1.3 竞争、锁定、死锁	300
13.1.4 等待与通知	303
13.2 并行	307
13.2.1 使用 subprocess 模块	307
13.2.2 使用 multiprocessing 模块	309
13.3 重点复习	312
课后练习	313
第 14 章 高级主题	314
14.1 属性控制	315
14.1.1 描述器	315
14.1.2 定义 __slots__	318
14.1.3 __getattr__(), __setattr__(), __delattr__()	320
14.2 装饰器	321
14.2.1 函数装饰器	321
14.2.2 类装饰器	324
14.2.3 方法装饰器	327
14.3 Meta 类	328
14.3.1 认识 type 类	328
14.3.2 指定 metaclass	330
14.3.3 __abstractmethods__	332
14.4 相对导入	333
14.5 重点复习	335
课后练习	336
附录 A venv	337
附录 B Django 简介	339

第 1 章

Python 起步走

学习目标

- 选择 2.x 还是 3.x
- 初识 Python 资源
- 认识 Python 的实现
- 建立 Python 环境

```
def filter(predicate, it):
    result = []
    for elem in it:
        if predicate(elem):
            result.append(elem)
    return result

C:\workspace\pdb_demo>python -m pdb filter_demo2.py
> filter_demo2.py(1)<module>()
-> def filter(predicate, it):
    1:     """Filter the elements of the iterable 'it' that satisfy the predicate 'predicate'."""
    2:     result = []
    3:     for elem in it:
    4:         if predicate(elem):
    5:             result.append(elem)
    6:     return result
```

```
1 -> def filter(predicate, it):
```

```
2     result = []
```

```
3     for elem in it:
```

```
4         if predicate(elem):
```

```
5             result.append(elem)
```

```
6     return result
```

```
8     def len_greater_than(num):
```

```
9         return len(element) > num
```

```
10     return len_greater_than(num)
```

```
11     return len_greater_than(num)
```

(Pdb)

```
def filter(predicate, it):
    """Filter the elements of the iterable 'it' that satisfy the predicate 'predicate'."""
    result = []
    for elem in it:
        if predicate(elem):
            result.append(elem)
    return result
```

```
def filter(predicate, it):
    """Filter the elements of the iterable 'it' that satisfy the predicate 'predicate'."""
    result = []
    for elem in it:
        if predicate(elem):
            result.append(elem)
    return result
```

Debugger Console

1.1 认识 Python

Python 诞生于 1991 年，至今足足有二十几个年头了，算是一门“古老的”程序设计语言。Python 经过这么长时间的发展，现在要学习 Python，究竟要先认识些什么，接下来将为读者详细介绍。

1.1.1 Python 3 的诞生

正因为 Python 是一门古老的程序设计语言，它的应用极为广泛，在系统管理、科学计算、Web 应用程序、嵌入式系统等各个领域都可以看到 Python 的踪迹。然而，本书并不打算从它的诞生开始谈起，如果读者有兴趣，可以看看百度百科或者维基百科上的“Python¹”词条。不过，就这个时间点来说，读者关心的应该不是 Python 的诞生，而是 Python 3 的发布。

时光暂且回到 2008 年 12 月 3 日，新出炉的 Python 3.0（也被称为 Python 3000 或 Py3K）中增添了许多人引颈期盼的新功能，其中最引人注目的是 Unicode 的支持，将 str/unicode 进行了整合，并明确地提供了另一个 bytes 类型，解决了许多人处理字符编码的问题。然而，其他语句与链接库方面的变更，也破坏了其向后的兼容性，导致许多基于 Python 2.x 的程序无法直接在 Python 3.0 的环境中运行。

提示>>>

在谈论这段历史的过程中，难免会出现一些专有名词，如果读者不清楚这些名词，先别担心，看过各章节的内容之后，回头再来看这些介绍，就会明白这些名词的意义。

对程序设计语言而言，破坏向后兼容性是一条危险的道路，历史上少有程序设计语言走这条路还能获得成功。许多程序设计语言在小心翼翼地推出新版的同时，兼顾向后兼容，代价往往就是语言越来越臃肿，有时想要吸收一些在其他程序设计语言中看似不错的特性，又为了保证符合向后的兼容性，结果总会将这类特性做些畸形的调整。特性越来越多，就会使得在处理一件任务时，错误与正确的做法越来越多，且并存于语言之中。

从这个层面来看，Python 3.0 选择破坏兼容性基本上是可以理解的。而 Python 3.0 演进的指导原则正是“将处理事情的老方法删除，以减少特性的重复”，这也符合“PEP 20²”的规范，也就是 Python 哲学（The Zen of Python，或 Python 禅学）中“做事时应该只有一种（也许是唯一）明确的方式”的原则。

然而，正如先前谈到的，Python 的应用极为广泛，以往累积起来的链接库等庞大资源，并不可能一朝一夕就升级到与 Python 3.0 兼容，因此在开发新的程序时，开始有人问“我要用 Python 2 还是 Python 3？”打算开始学习 Python 的人们也在问“我要学 Python 3 还是 Python 2？”

在 Python 3.0 发布后没多久，答案通常会“学习 Python 2.x，因为许多链接库还不支持 Python

¹ 百度百科的<Python>词条：<http://baike.baidu.com/>；维基百科的〈Python〉词条：zh.wikipedia.org/wiki/Python

² PEP 20：www.python.org/dev/peps/pep-0020/

3.0!”然而，随着时间的推移，答案渐渐地变得难以选择，许多介绍 Python 的入门文档或书籍，也不得不同时介绍 Python 2 与 Python 3 两个版本；尽管有“2to3”¹这个工具声称可以将 Python 2 的程序代码转换为 Python 3，但是它也不能发现所有问题；渐渐地，甚至有了“Python 3 is killing Python”²这类文章出现，预测着 Python 社区将会分裂，甚至预测现有的拥护者也可能会离开 Python。

1.1.2 从 Python 3.0 到 3.5

尽管破坏了向后兼容性的程序设计语言多半不会有什么好结果，然而，就这几年来从 Python 3.x 与 2.x 的发展来看，过程与那些失败了的程序设计语言不太一样。

官方的推动

首先，Python 本身以每隔一年左右推出一个 3.x 版本推进着，过程也并不是官方一厢情愿地推进，而是不断地倾听着 Python 社区的声音，不断地为兼容转换做出努力。表 1.1 为 Python 3.0 到 3.5 发布的日期。

表 1.1 Python 3.0 到 3.5 发布的日期

版本	发布的日期
Python 3.0	2008/12/03
Python 3.1	2009/06/27
Python 3.2	2011/02/20
Python 3.3	2012/09/29
Python 3.4	2014/03/16
Python 3.5	2015/09/13

举例来说，如果想在 Python 2 中就开始使用 Python 3.x 的一些特性，可以试着通过 `from __future__ import` 来使用想使用的模块，例如最基本的 `from __future__ import print_function` 就可以开始使用 Python 3.x 中的 `print()` 函数（Function），以兼容方式来编写输出语句。

Python 官方的“Python 2 or Python 3”³也整理了许多兼容转换的相关资源。其中指出，Python 3.0 的一些比较不具破坏性的特性，反向移植（backport）到了 Python 2.6 中，Python 3.1 的特性反向移植到了 Python 2.7 中；也会反过来从 2.x 移植至 3.x。例如，在 Python 3.3 中又支持了 `u"foo"` 来表示 unicode 字符串，`b"foo"` 来表示 byte 字符串，兼容性同时在 2.x 与 3.x 之间推进着，试着让不同版本的语句及其语法格式有更多交集。

Python 3.x 本身也不断地吸纳 Python 社区的经验，举例来说，Python 3.3 中包含了 `venv` 模块，相当于过去社区用来建立虚拟环境的 `virtualenv` 工具；Python 3.4 本身就包含了 `pip`，这是过去 Python 社区中，建议用来安装 Python 相关模块的工具；Python 3.5 进一步纳入了 Type hints，尽管 Python 本身是个动态数据类型语言，然而，Type hints 特性有助于静态分析、重构、运行时刻的类型检查，

¹ Automated Python 2 to 3 code translation: docs.python.org/3.0/library/2to3.html

² Python 3 is killing Python: blog.thezerobit.com/2014/05/25/python-3-is-killing-python.html

³ Python 2 or Python3: wiki.python.org/moin/Python2orPython3

这对大型项目开发有显著帮助，而且对现有程序代码不会有影响。

社区的接纳

尽管 Python 3.x 本身不断在兼容性与新特性上发布好消息，但是 Python 社区不买账也是徒劳无功，所幸实际情况并非如此。

在 Python 官方的持续推动之下，许多基于 Python 2.x 的链接库或框架在这段期间不断地往 Python 3.x 移植，例如 Web 快速开发框架 Django，在笔者著书期间，新版本已经支持至 Python 3.4，官方 Tutorial 也是基于此版本而改写的，一些科学运算软件包，像 Numpy、SciPy 等也有了支持 Python 3.x 的版本。

想要知道其他链接库是否支持 Python 3.x，有个“PYTHON 3 WALL OF SUPERPOWERS¹”可以作为不错的参考资料，其中列出了 200 个链接库，174 个标示为绿色，表示支持 Python 3.x，在 2014 年左右，仍有 35 个被标示为不支持 Python 3.x 的红色，当前只剩下 26 个链接库是红色状态。

一个现实的问题是，新系统开发时究竟要基于 Python 2.x 还是 Python 3.x？最好的方式是编写出能同时兼容 2.x 与 3.x 版本的程序代码，除了要建立良好的程序代码编写习惯之外，Python 社区中还有着 six²这类的软件包，可以作为编写出兼容 Python 2.7 和 Python 3.x 版本的程序代码的基础。

另一方面，由于 Python 2.7 是 Python 2 的最后一个版本（不会再有 Python 2.8），且预计只支持到 2020 年，在停止支持之前的这段日子，也不再加入新特性，只会针对程序中的错误（bug）或安全问题进行修正，因此除许多链接库已经在进行移植之外，有些操作系统也开始进行相对应的移植工作。

像 Linux 系统，目前多半同时默认加载了 Python 2.x 与 Python 3.x。以 Ubuntu 为例，从 13.04 之后的版本开始就默认加载了 Python 3.x，Ubuntu 也在持续去除系统中对 Python 2.x 的依赖，在其未来的计划中，希望有朝一日能够全面采用 Python 3.x，而不再默认加载 Python 2.x。

当然，一定还会有人死守着 Python 2.x，宣称未来绝不会支持 Python 3.x 的软件包、链接库或应用程序。身为 Python 开发者，将来也可能遇到必须面对这些链接库的时候，然而更重要的是，无论现阶段个人偏好如何，在遇到类似“我要学习 Python 2.x 还是 Python 3.x？”的问题时，答案不应只是简单的“学习 Python 2.x，因为许多链接库还不支持 Python 3.x”，而是应该针对自己或客户的需求，进行全面性的调查，就像在选择一门程序设计语言，或者是调查某个链接库是否可以采用时，必须进行多方面的考虑，像了解其更新（Update）的时间、更新日志（Changelog）、修正问题（Issue）的频度、作者身份等。

当然，从 Python 2.7 终究会在 2020 年停止支持，以及官方在 Python 3.x 上的推动和 Python 社区支持这两个方面来看，未来 Python 的生态圈，应会持续接纳 Python 3.x，因而就学习 Python 来说，可以先学 Python 3.x，因为有了 Python 3.x 的基础，将来若有必要面对或学习 Python 2.x，就不会是件难事。

¹ PYTHON 3 WALL OF SUPERPOWERS: python3wos.appspot.com/

² six: pypi.python.org/pypi/six

1.1.3 初识 Python 的社区资源

认识一门程序设计语言，不能只是学习语言的语法，更要逐步深入了解语言背后的社区与文化。对于 Python 这门语言来说，这点更为重要，最好的方式就是从认识语言创建者开始，了解语言设计的理念，接着从社区网站出发，搜索更多可以了解的资源并积极参与社区的活动。

Python 之父

使用 Python 可别不认识 Python 的创建者 Guido van Rossum¹，Guido 是首位享有 BDFL 封号的开放源码（或简称开源）软件创建者，BDFL 全名为 Benevolent Dictator For Life，中文常译为“仁慈的独裁者”，意思是拥有这类称号的开源软件创建者，对社区仍持续关注，在必要时能对社区中的意见与争议提出想法并作出最后的裁决。Python 3.x 得以持续推进就是一个例子，因为 Python 3.x 正是 Guido van Rossum 的最爱，没有他的坚持，或许 Python 3.x 难以有今日的接受度。

提示 >>>

在 Guido van Rossum 的 Google+ 专页上，就曾经张贴过一则真实的笑话（<https://goo.gl/S43JBx>），有猎头公司的人写信给 Guido，说通过 Google（谷歌）搜索看过他的简历，觉得他在 Python 上似乎极为专业，想介绍一个 Python 开发方面的职位给他。

Guido 在 2005 年至 2012 年曾受雇于 Google 公司，大半时间在维护 Python 的开发，2013 年之后离开 Google 进入 Dropbox，可以在官方个人页面找到他，上面也会告诉你 Guido van Rossum 究竟怎么发音。

Python 软件基金会

Python Software Foundation²常简称为 PSF，主要任务是推广、维护与促进 Python 程序设计语言的发展，同时也支持和协助全球各地各种各样 Python 程序设计师与 Python 社区的成长。PSF 是非营利组织，持有 Python 程序设计语言的知识产权。

Python 改进提案

Python Enhancement Proposals³常简称为 PEPs，Python 的改进多是由 PEP 流程主导，PEP 流程会收集来自 Python 社区的意见，为将来打算加入 Python 的新特性提出文档提案，重要的 PEP 会经由 Python 社区与 Guido 审阅与评估，决定是否成为正式的 PEP 文档。

因此 PEP 文档本身说明了它对 Python 的改变，以及实现特性时应遵守的标准，在刚开始认识 Python 时，有几个重要的 PEP 文档是必须认识的，像 PEP 1、PEP 8、PEP 20、PEP 257 等，如表 1.2 所示。

¹ Guido van Rossum: www.python.org/~guido/

² Python Software Foundation: www.python.org/psf/

³ Python Enhancement Proposals: www.python.org/dev/peps/

表 1.2 初学应认识的 PEP 文档

文档	说明
PEP 1	PEP 的作用与执行准则，说明什么是 PEP、PEP 的类型、提案方式等
PEP 8	Python 的程序代码风格，包括程序代码的编排、命名、注释等风格指引，想编写出有 Python 风格的程序代码，必定要参考的一份文档
PEP 20	Python 哲学（The Zen of Python，原意为 Python 禅学），编写 Python 时的精神指引，或者说是金玉良言，像“美丽优于丑陋”（Beautiful is better than ugly）、“明确胜于隐晦”（Explicit is better than implicit，即显式胜于隐式）等
PEP 257	编写 Docstrings 时的惯例，Docstrings 是可内建于 Python 程序中的帮助文档字符串，第 6 章会加以介绍

Python 研讨会

全世界都有 Python 用户，这些用户会在各地举办大大小小的 Python 研讨会（Python Conference），如果想要知道世界各地的研讨会信息，可以从 PyCon¹网站开始，它列出了全球各地 Python 研讨会的网址、活动日期等信息。

在 PyCon 网站上，你可以找到 "PyCon China" in China 链接，单击链接之后，就可以看到中国 Python 社区关注的重要研讨会信息。

Python 用户群

除了研讨会之外，Python 用户会举办周期性的聚会，可以在“LocalUserGroups²”上找到全球各地的 Python 用户聚会信息，以中国来说 Python 用户群上面记录的周期性聚会信息包含 Python Chinese User Group（Python 中国用户群组，即 CPUG Wiki 网站）、BPUG（北京 Python 用户群——CPUG 的子群）、ZEUUX（一个含有 Python 用户群的自由软件社区）。

1.2 建立 Python 环境

在这本书中，将会使用 Python 3.5 作为开发和运行的环境，然而现今这个时间点，操作系统的选择上多元化了，基于篇幅考虑，在介绍如何安装 Python 的这一节中，只会以在 Windows 操作系统中建立 Python 环境进行介绍，因为相对来说，Windows 的用户可能在建立程序设计相关环境上缺少经验，因而会需要较多这方面的协助。

1.2.1 Python 的实现

在介绍如何安装 Python 之前，先来认识几个 Python 的实现产品。能执行 Python 语言的软件产品不少，接下来介绍几个主要的 Python 实现产品。

¹ PyCon: www.pycon.org

² LocalUserGroups: wiki.python.org/moin/LocalUserGroups

CPython

CPython 是 Python 官方的参考实现，一般提到安装 Python，如果没有特别声明，多半是指安装 CPython。顾名思义，它是以 C 语言编写的软件产品，提供了 Python 软件包（Package）与 C 扩展模块的最高兼容性，作者安装的 Python 环境就是 Windows 版本的 CPython。

Python 是解释型语言，不过并非每次都从源码直接解释后再执行，CPython 会将源码解析为字节码（Bytecode），然后再由虚拟机执行，也就是之后再次执行同一程序，如果源码文件被检测到没有被更改，就不会再对源码重头进行语法和语义解析等操作，而是从字节码开始直接解释，以加快解释执行的速度。

PyPy

从 PyPy¹名称上来看，是用 Python 实现的 Python。正确地说，是使用 RPython(Restricted Python)来实现的 Python。RPython 不是完整的 Python，是 Python 的子集，不过 PyPy 可以执行完整的 Python 语言，运行速度比 CPython 快，目的在于改进 Python 程序的执行性能，同时追求与 CPython 的最大兼容性。

对 Python 3.x 的支持来说，PyPy 是个指标性代表，在编著本书期间，它有支持 Python 2.7.10 与 Python 3.2.5 的版本。

Jython

Jython²是用 Java 实现的 Python，会将 Python 程序代码转译为 Java 的字节码，因此可让使用 Python 语言编写的程序运行于 Java 虚拟机（Java Virtual Machine, JVM）上，既然可以运行在 JVM 上，也就能导入、引用 Java 的相关链接库，因而得以运用 Java 领域中的各种资源。

在编写本书时，Jython 的最新版本是 2.7，而 Jython 的主要开发者之一 Frank Wierzbicki 曾表示，在 Jython 2.7 之后，会开始认真地着手 Jython 3。

IronPython

IronPython³是可与 .NET 平台结合的 Python 开放源码的实现，可以使用 .NET Framework 链接库，也让 .NET 平台上的其他语言易于使用 Python 链接库。

IronPython 的创建者 Jim Hugunin 同时也是 Jython 的创建者。IronPython 3⁴是以支持 Python 3.x 为目标的一个项目。

¹ PyPy: pypy.org/

² Jython: www.jython.org/

³ IronPython: ironpython.net/

⁴ IronPython 3: github.com/IronLanguages/ironpython3

1.2.2 下载与安装 Python 3.5



要下载 Python，请连接到 Python 官方网站 www.python.org，在首页的“Downloads”菜单中会自动检测操作系统，直接列出可下载的安装文件，如图 1-1 所示。

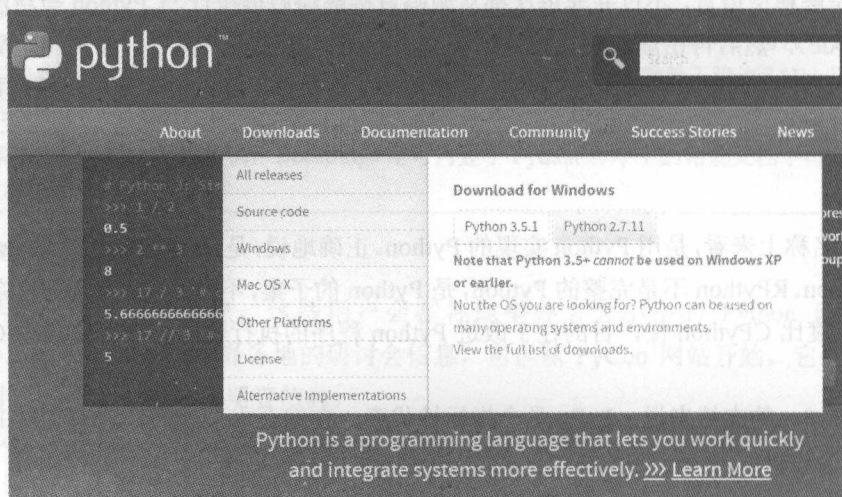


图 1-1 下载 Python 3.5

注意

从图 1-1 上可以看到，Python 3.5 或之后的版本无法在 Windows XP 上使用，本书将在 Windows 10 中示范安装过程。

Python 发布时的版本号采用的是 major.minor.micro 的形式，也就是主版本号、次版本号与小版本号。主版本号只有在语言本身确实有重大变革时才会递增，从 Python 2 变成 Python 3 就是个例子；次版本号是在增加了重要特性，但不至于影响整个语言层面时递增，基本上约一年多推出一次，像 Python 3.0 至 3.5 这样的过程；小版本号基本上每隔一段时间就会发布，用以修正程序错误（bug）之类的问题。

Python 3.5 是在 2015 年 9 月 13 日发布的，而图 1-1 看到的 Python 3.5.1 是 2015 年 12 月 6 日发布的第一个程序错误（bug）修正版本，读者之后看到的版本，也会是 Python 3.5.x 这样的版本号形式。

提示

对 Python 的版本语义有更多的兴趣吗？这些规范在 PEP 0440 文档之中：
www.python.org/dev/peps/pep-0440/

对于 Windows 用户，单击图 1-1 中的“Python 3.5.1”按钮，就会下载一个 python-3.5.1.exe 文件，因为是从网络下载可执行文件，根据 Windows 中的安全设置等级的不同，你可能必须用鼠标右击该文件，弹出快捷菜单后，在“常规”选项卡进行“解除锁定”的操作，之后才能进一步运行

此安装程序，如图 1-2 所示。

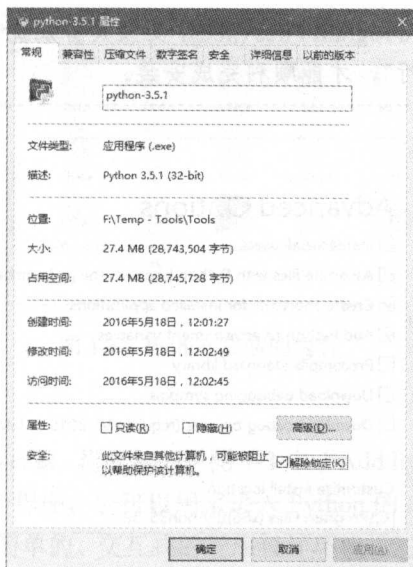


图 1-2 解除锁定

如图 1-2 所示勾选“解除锁定”并单击“确定”按钮后，再次用鼠标右击 `python-3.5.1.exe`，然后选择弹出菜单中的“以管理员身份运行”进行安装，就会看到图 1-3 所示的安装起始画面。

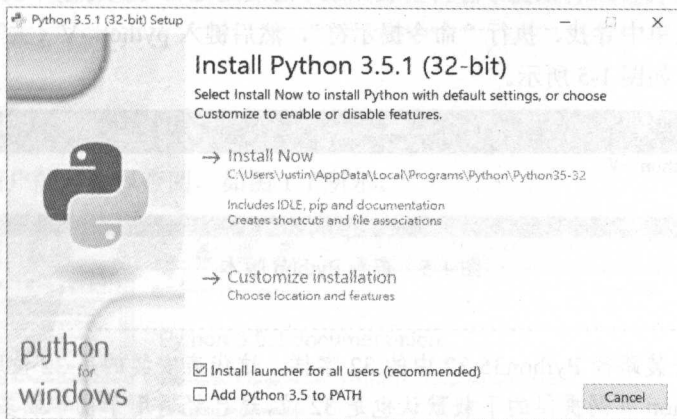


图 1-3 安装起始画面

当用户在“命令提示符”（又称 Console，中文常称为控制台）中键入某个命令时，操作系统会查看 PATH 环境变量设置的文件夹位置中是否能找到指定的脚本文件，因此请勾选“Add Python 3.5 to PATH”，这样就不用再去手工设置 PATH 环境变量，对于初学者比较方便。

在图 1-3 中可以看到，默认用来安装 Python 3.5 的路径是用户文件夹下的 `AppData\Local\Programs\Python\Python35-32` 文件夹，这文件夹路径太冗长了，若想改变这个路径，可以单击“Customize installation”，然后在下一个画面中直接单击“Next”按钮，就会出现可以进行修改的字段。例如，读者可以将之安装到 `C:\Program Files (x86)\Python35-32` 中，如图 1-4 所示。

注意>>>

如果选择将 Python 安装至 C:\Program Files (x86)，必须具备系统管理员身份，也就是先前提及的必须“以管理员身份运行”，才能顺利完成安装。

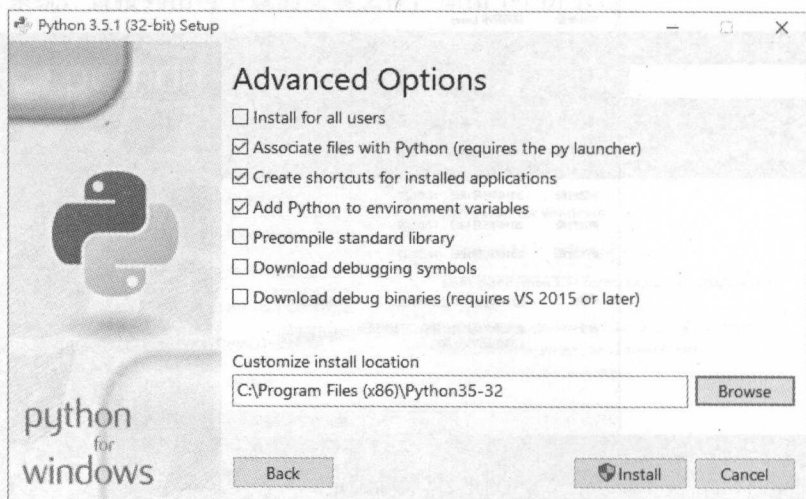


图 1-4 自定义安装文件夹

接着就只要单击“Install”按钮，静待安装完成，想确定是否可执行基本的 python 命令，可以在 Windows 的程序菜单中寻找、执行“命令提示符”，然后键入 `python -V`（大写的 V），这时会显示出 python 的版本，如图 1-5 所示。

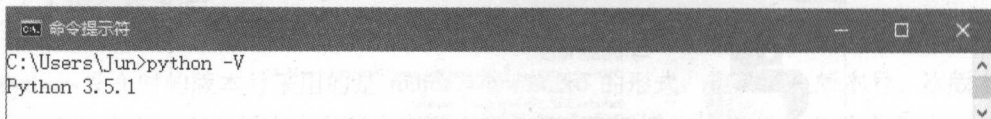


图 1-5 查看 Python 版本

提示>>>

你可能会留意到安装路径 Python35-32 中的 32 字样，这代表安装的是 32 位版本。即使操作系统是 64 位，Python 官网提供的下载默认也是 32 位，这已经适用于初学或多数的应用程序。如果未来基于某些考虑（比如想使用更多的内存），打算采用 64 位版本，也可以在官方网站下载。例如，Python 3.5.1 可在 www.python.org/downloads/release/python-351/ 中找到 64 位的版本。

1.2.3 认识安装的内容

那么你到底安装了哪些东西呢？无论是默认安装到用户文件夹中的 `AppData\Local\Programs\Python\Python35-32`，还是如图 1-4 所示安装到 `C:\Program Files (x86)\Python35-32`，现在都请打开对应的文件夹，认识几个重要的安装内容，如图 1-6 所示。

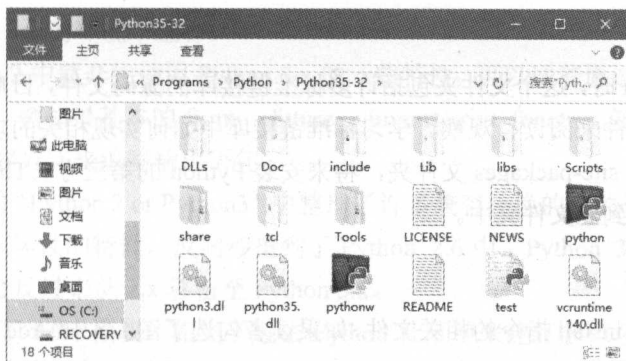


图 1-6 Python 安装的内容

python 指令

在图 1-6 中可以看到 `python.exe`，若你勾选了图 1-3 中 “Add Python 3.5 to PATH”，那么图 1-5 下达 `python` 指令时就会运行这个程序，它可以用来进入 Python 的 REPL (Read-Eval-Print Loop，即交互编程) 环境，也就是一个简单的、交互式的程序设计环境，之后的章节会介绍 REPL。`python` 指令也用来运行 Python 源码或模块，我们将会频繁地使用这个指令。

提示

你应该还看到了 `pythonw.exe`，如果开发了桌面图形界面应用程序，那么使用这个指令可以不出现控制台画面，也就是在 Windows 中可以不出现“命令提示符”运行的界面。

Doc 文件夹

在 Windows 版本的 Python 中，这个文件夹中提供了一个 `python351.chm` 文件，它包含了许多 Python 文档，方便用户随时选择查阅，如图 1-7 所示。

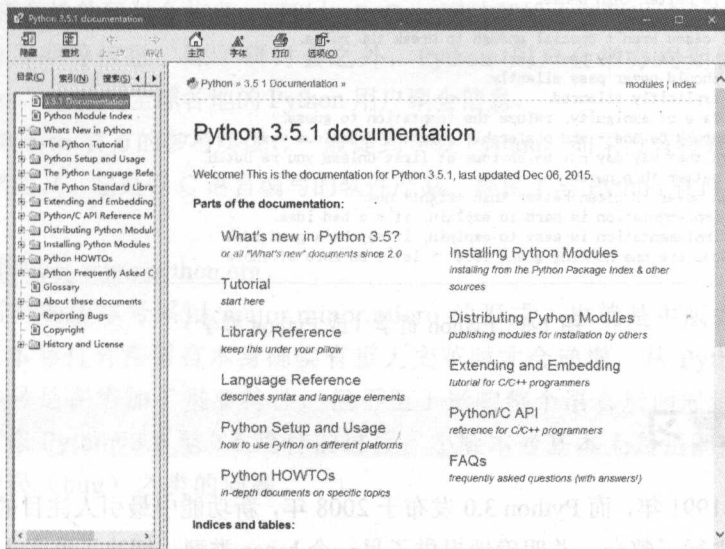


图 1-7 Python 帮助文件

Lib 文件夹

在刚安装完 Python 时, 这个文件夹包括许多标准链接库的源码文件, 日后有能力且有兴趣时, 可以试着找几个源码文件来阅读、观察、学习标准链接库中如何实现相关的功能。

这个文件夹中有个 `site-packages` 文件夹, 将来安装 Python 的第三方 (Third-party) 链接库时, 通常会将相关文件存放到此文件夹中。

Scripts 文件夹

包含了 `pip` 与 `easy_install` 指令的相关文件, 如果读者勾选了图 1-3 中“Add Python 3.5 to PATH”, 那么 PATH 环境变量中也会包含 Scripts 文件夹, 因此在“命令提示符”程序中也可以执行 `pip` 与 `easy_install` 指令。

Tools 文件夹

一些范例程序代码, 以及使用 Python 编写的工具程序, 例如其中的 `scripts` 文件夹就包括了 `2to3.py` 文件, 用来将 Python 2 的源码转换成 Python 3 的源码。

进入 Python 的准备工作就先到这里了, 从下一章开始, 就可以正式进行 Python 程序的编写了。如果读者意犹未尽, 那么试着在“命令提示符”程序中键入 `python -c "import this"`, 之后会出现一段文字 (见图 1-8), 也就是 1.1.3 小节提过的 Python 哲学 (或 Python 哲学), 试着先从文字中品味一下 Python 的精神吧!

```
C:\Users\Jun>python -c "import this"
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

图 1-8 Python 哲学 (或 Python 禅学)

1.3 重点复习

Python 诞生于 1991 年, 而 Python 3.0 发布于 2008 年, 新功能中最引人注目的是 Unicode 的支持, 将 `str/unicode` 进行了整合, 并明确地提供了另一个 `bytes` 类型, 解决了许多人处理字符编码的问题。然而, 其他语法与链接库方面的变更也破坏了向后的兼容性, 导致许多基于 Python 2.x 的程

序无法直接在 Python 3.0 的环境中运行。

如果想在 Python 2 中就开始使用 Python 3.x 的一些特性，可以试着通过 `from __future__ import` 来使用想使用的模块，例如最基本的 `from __future__ import print_function` 就可以使用 Python 3.x 中的 `print()` 函数，以兼容方式来编写输出语句。

在 Python 官方的“Python 2 or Python3”中整理了许多兼容转换的相关资源。其中指出，Python 3.0 的一些比较不具破坏性的特性，反向移植到了 Python 2.6 中，Python 3.1 的特性反向移植到了 Python 2.7 之中；也会反过来从 2.x 移植至 Python 3.x。

Python 3.3 包含了 `venv` 模块，相当于过去 Python 社区用来建立虚拟环境的 `virtualenv` 工具；Python 3.4 本身就包含了 `pip`，这是过去在 Python 社区中，建议用来安装 Python 相关模块的工具；而 Python 3.5 进一步纳入了 Type hints。

想要知道其他链接库是否支持 Python 3.x，有个“PYTHON 3 WALL OF SUPERPOWERS”，可以作为不错的参考资料。

Python 2.7 是 Python 2 的最后一个版本（不会再有 Python 2.8 了），且预计只支持到 2020 年。

Python 的创建者 Guido van Rossum，是首位享有 BDFL 封号的开放源码软件创建者，BDFL 全名为 Benevolent Dictator For Life，中文常翻为“仁慈的独裁者”，意思是拥有这类称号的开放源码软件创建者，对社区仍持续关注，在必要时能对社区中的意见与争议提出想法，并作出最后的裁决。

Python Software Foundation 常简称为 PSF，主要任务为推广、维护与促进 Python 程序设计语言的发展，同时也支持协助全球各地各种各样 Python 程序设计师与 Python 社区的成长。PSF 是非营利组织，持有 Python 程序设计语言的知识产权。

Python Enhancement Proposals 常简称为 PEPs，Python 的改进多是由 PEP 流程主导，PEP 流程会收集来自 Python 社区的意见，为将来打算加入 Python 的新特性提出文档提案，重要的 PEP 会经由 Python 社区与 Guido 审阅与评估，决定是否成为正式的 PEP 文档。

如果想要知道各地的研讨会信息，可以从 PyCon 网站开始，它列出了全球各地 Python 研讨会的网址、活动日期等信息。除了研讨会之外，Python 用户会举办周期性的聚会，可以在“LocalUserGroups”上找到全球各地的 Python 用户聚会信息。

CPython 是 Python 官方的参考实现，一般提到安装 Python，如果没有特别声明，多半是指安装 CPython。顾名思义，它是以 C 语言编写的软件产品，提供了与 Python 软件包与 C 扩展模块的最高兼容性。

Python 官方网站是 www.python.org。

Python 发布时的版本号采用 `major.minor.micro` 的形式，也就是主版本号、次版本号与小版本号。主版本号只有在语言本身确实有重大变革时才会递增，从 Python 2 变成 3 就是个例子；次版本号是在增加了重要特性，但不至于影响整个语言层面时递增，基本上约一年多推出一次，像 Python 3.0 至 3.5 这样的过程；小版本号基本上是每隔一段时间就发布，用以修正程序错误（bug）之类的问题。

第 2 章

从 REPL 到 IDE

学习目标

- 使用 REPL
- 设置源码文件编码
- 基本模块与软件包管理
- 认识 IDE 的使用

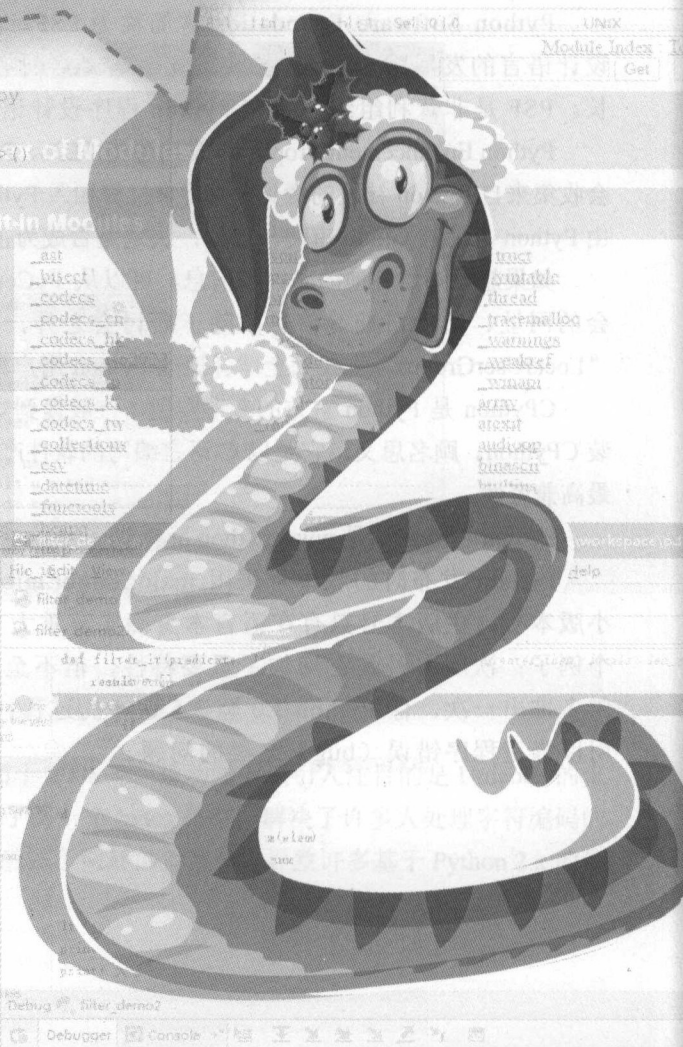
```
py
py x
es_it/predicate, lt)
a = []
len greater than num elem
store len greater than num
len greater than num
-> def filter_if(predicate, lt):
    result = []
    for elem in lt:
        if predicate(elem):
            result.append(elem)
    return result
C:\workspace\pdb_demo>python -m pdb filter_demo2.py
> c:\workspace\pdb_demo\filter_demo2.py(1)<module>()
>
-> def filter_if(predicate, lt):
    result = []
    for elem in lt:
        if predicate(elem):
            result.append(elem)
    return result
```

```
1 -> def filter_if(predicate, lt):
2     result = []
3     for elem in lt:
4         if predicate(elem):
5             result.append(elem)
6     return result
7
8 def len_greater_than(num):
9     def len_greater_than(elem):
10         return len(elem) > num
11     return len_greater_than
(Pdb)
```

Built-In Modules

ast
base64
codecs
codecs_cn
codecs_hk
codecs_iso8859
codecs_jp
codecs_kr
codecs_tw
collections
csv
datetime
functools

Module Index
Get



2.1 从 'Hello World' 开始

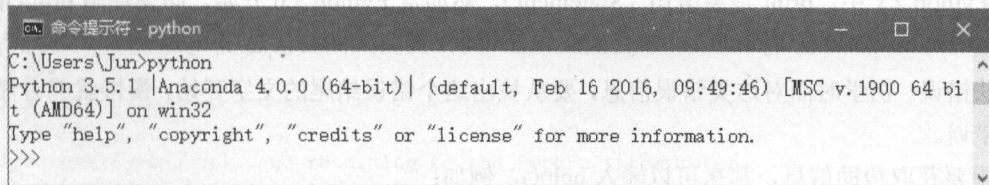
第一个'Hello World'的出现是在 Brian Kernighan 写的《A Tutorial Introduction to the Language B》书中（B 语言是 C 语言的前身），用来将'Hello World'文字显示在计算机屏幕上，自此之后，很多程序设计语言的教学文档或书籍上，无数次地将它当作第一个范例程序。为什么要用'Hello World'来作为第一个程序范例呢？因为它很简单，初学者只要键入几行简单的程序（甚至一行），就可以要求计算机执行并得到回馈：显示'Hello World'。

本书也要从显示'Hello World'开始，然而在完成这个简单的程序之后，千万要记得探索这个简单程序之后的种种细节，千万别过于乐观地以为，你想从事的程序设计工作就是如此容易驾驭。

2.1.1 使用 REPL

第一个显示'Hello World'的程序代码，我们在 REPL（Read-Eval-Print Loop）环境中进行（又称为 Python Shell），这是一个简单、交互式的程序设计环境。不过，虽然它很简单，但是在日后开发 Python 应用程序的日子里，读者会经常地使用它，因为 REPL 在测试一些程序片段的操作时非常方便。

现在启动“命令提示符”，直接输入 `python` 指令（不用加上任何参数），这样就会进入 REPL 环境，如图 2-1 所示。



```
命令提示符 - python
C:\Users\Jun>python
Python 3.5.1 [Anaconda 4.0.0 (64-bit)] (default, Feb 16 2016, 09:49:46) [MSC v.1900 64 bit (AMD64)] on win32
Type 'help', 'copyright', 'credits' or 'license' for more information.
>>>
```

图 2-1 Python 的 REPL 环境

现在来很快地编写一些小指令进行测试，首先做些简单的加法运算吧！从输入 `1+2` 之后按【Enter】键开始。

```
>>> 1 + 2
3
>>> _
3
>>> 1 + _
4
>>> _
4
>>>
```

一开始执行了 `1+2`，显示结果为 3，`_` 代表了互动环境中上一次的运算结果，方便用户在下次运算中直接引用上一次的运算结果。

```
print(...)
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

...略

C:\Users\Jun>
```

提示>>>

在 Python 官方网站 (www.python.org) 首页, 也提供了一个互动环境, 临时要测试一个程序片段, 又不想安装 Python 或找个装有 Python 的计算机时, 启动浏览器就可以使用了, 如图 2-2 所示。

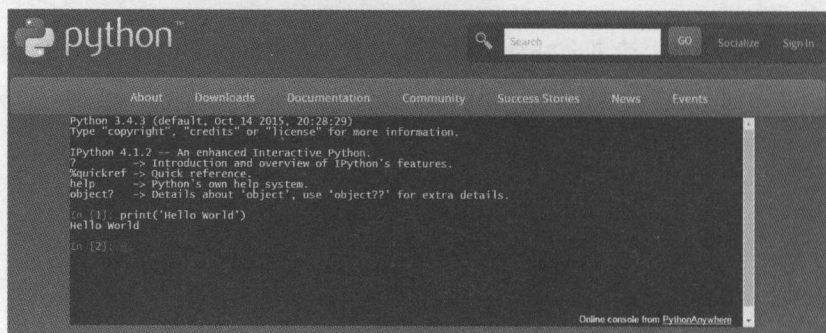


图 2-2 Python 官网上提供的在线 REPL 环境

2.1.2 编写 Python 源码



我们总是要打开一个源码文件, 正式一点地编写程序吧? 在正式编写程序之前, 请先确定自己可以看到文件的扩展名, 在 Windows 下默认不显示扩展名, 这会造成重新命名文件时的困扰, 如果当前在“文件资源管理器”下无法看到扩展名, Windows 7 下请依次选择“组织→文件夹和搜索选项”, 在 Windows 8 或 10 中则可以依次选择“查看→选项”, 之后切换至“查看”标签, 取消勾选“隐藏的项目”, 同时确保勾选了“文件扩展名”选项, 如图 2-3 所示。

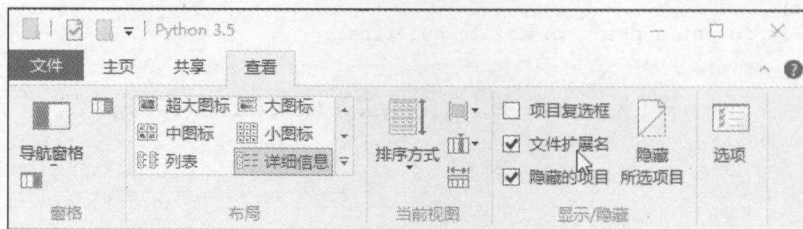


图 2-3 选择显示出“文件扩展名”

接着选择一个文件夹来编写 Python 源码文件, 本书都是在 C:\workspace 文件夹中编写程序, 请新建“文本文件”(也就是.txt 文件), 并重新命名文件为“hello.py”, 由于将文本文件的扩展名从.txt 改为.py, 系统会询问是否更改扩展名, 请确认更改。如果是在 Windows 中第一次安装 Python, 而且按照本章之前介绍的方式安装, 那么会看到文件图标变换为以下样式, 这个图标上的吉祥物是两只小蟒蛇(因为 Python 也有蟒蛇之意), 如图 2-4 所示。

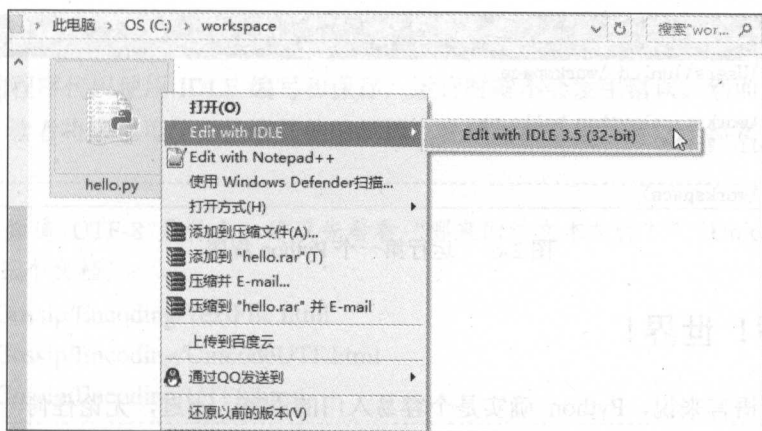


图 2-4 更改扩展名为.py 后的图标

提示 >>>

尽管 Python 有蟒蛇之意，不过 Guido van Rossum 曾经表示，Python 这个名称其实是取自他热爱的 BBC 著名喜剧电视剧《Monty Python's Flying Circus》，Python 官方网站的 FAQ 也记录着这件事。

docs.python.org/3/faq/general.html#why-is-it-called-python

这个 FAQ 的下一个还是很幽默地列出了 Do I have to like “Monty Python’s Flying Circus”? 这个问题，答案是 No, but it helps.

如图 2-4 中可以看到，如果在.py 文件上右击鼠标，可以选择“Edit with IDLE”选项来打开这个文件进行编辑，IDLE 是 Python 官方内建的编辑器（本身也是使用 Python 编写而成），这会比使用 Windows 内建的记事本编写 Python 程序要好一些，读者可以如图 2-5 所示来编写程序代码。

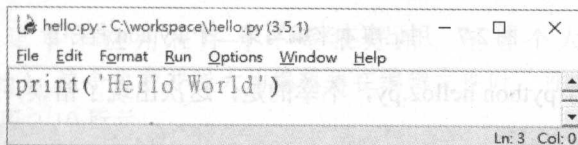


图 2-5 使用 Python IDLE 来编写第一个 Python 程序

很简单的一个小程序，只是使用 Python 的 print() 函数输出文字，执行时 print() 函数默认会在控制台（Console）显示要输出的文字。

接着选择菜单“File/Save”选项来保存文件。虽然可以直接在 IDLE 中选择“Run/Run Module”选项来启动 Python 的 REPL 来运行程序。不过，下面要使用“命令提示符”程序。请启动“命令提示符”程序，工作文件夹切换至 C:\workspace（执行命令 cd c:\workspace），然后如图 2-6 所示使用 python 指令运行程序。

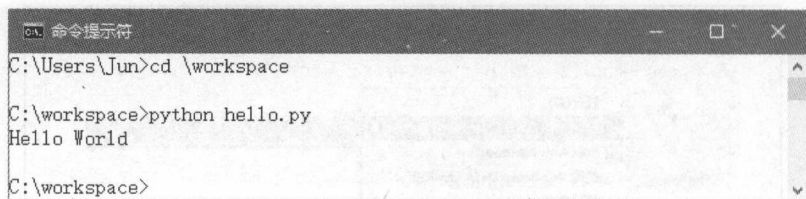


图 2-6 运行第一个 Python 程序

2.1.3 哈啰！世界！

就程序设计语言来说，Python 确实是个容易入门的语言。不过，无论任何一个领域都请记住“事物的复杂度不会凭空消失，只会从一个事物转移到另一个事物”，在程序设计这个领域也是如此。如果纯粹只是想“营销”Python 这门语言给读者，介绍完刚才的 Hello World 程序后，笔者就可以开始“歌颂”Python 语言的美好了。

Python 有其“美好”的一面，实际上也有其会面临的难题，如果读者将来打算发挥 Python 更强大的功能，或者需要解决更复杂的问题，就需要进一步深入探索 Python。在本书之后的章节，会谈到 Python 一些比较深入的议题。

至于现在，作为中文世界的开发者，想稍微触碰一下复杂度，除了显示 Hello World 之外，不如再来试试显示“哈啰！世界！”如何？请创建一个 hello2.py 文件，这次不要使用 IDLE，直接使用 Windows 的“记事本”来编写这个程序，如图 2-7 所示。

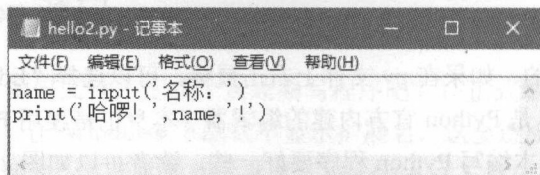


图 2-7 用记事本来编写第二个 Python 程序

接着在控制台中运行 `python hello2.py`，不幸的是，这次出现了错误，如图 2-8 所示。

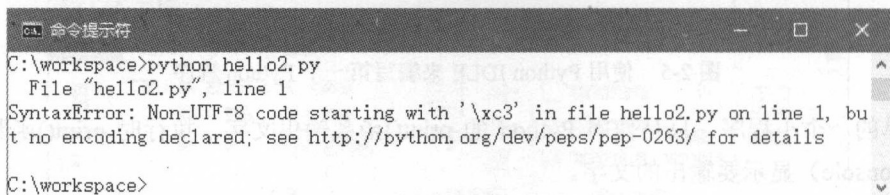


图 2-8 运行出错了

UTF-8

错误的信息中显示 `SyntaxError`，也就是语法错误，原因在于 Python 3 之后，python 解释器预期的源文件的编码必须是 UTF-8（Python 2.x 预期的是 ASCII）。然而，Windows 简体中文版中的记事本默认的编码是 MS936（兼容于汉字国标码 GBK），两者对于汉字字符的字节编码方式并不相同，python 解释器如果看到无法解释的字节，就会发生了 `SyntaxError`。

UTF-8 是目前很流行的文字编码方式，现在不少文本编辑器默认会使用 UTF-8，像刚才使用的 IDLE，相同的程序代码使用 IDLE 编写和保存，运行时就不会发生错误。然而问题在于，许多人不使用 IDLE，读者将来也可能会换用其他编辑器，在使用前必须了解这个事实。

提示

如果读者尚不知道 UTF-8 是什么，建议先看看“哪来的纯文本文件？”“Unicode 与 UTF”“UTF-8”这三个文档。

openhome.cc/Gossip/Encoding/TextFile.html

openhome.cc/Gossip/Encoding/UnicodeUTF.html

openhome.cc/Gossip/Encoding/UTF8.html

Windows 的记事本可以在“另存为”时选择文字编码为 UTF-8，如图 2-9 所示，这是解决问题的一个方式。

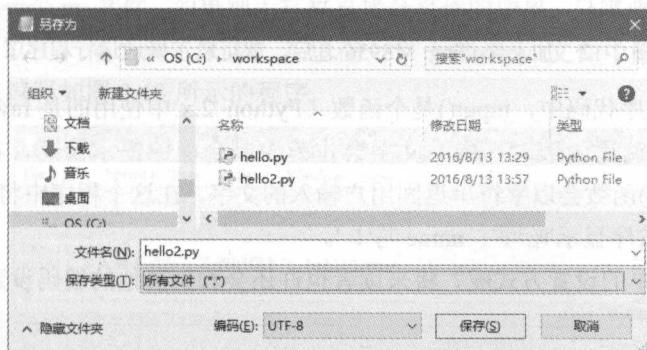


图 2-9 记事本可在“另存为”时选择文字编码为 UTF-8

提示

Windows 中内建的记事本应用程序并不是很好用，笔者个人习惯用 Notepad++ (notepad-plus-plus.org)，这个免费并强大的编辑器在编辑文件时，可以直接在菜单“编码”中选择文字编码，如图 2-10 所示。



图 2-10 设置 Notepad++ 的文字编码方式

设置源码的编码

若不想将文件的文字编码设置为 UTF-8，另一个解决方式是在源码的第一行使用注释方式来设置编码。最简单的设置方式如图 2-11 所示。

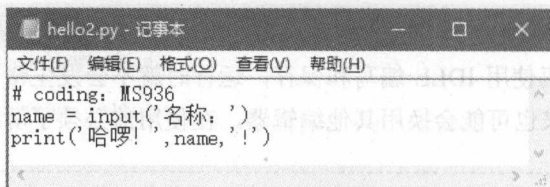


图 2-11 设置 Python 源码文件的编码

在 Python 源码文件中，# 开头代表这是一行注释，# 之后不会被当成是程序代码的一部分，如图 2-11 所示加上 #coding: MS936 之后，就可以正确地运行程序了，如图 2-12 所示。

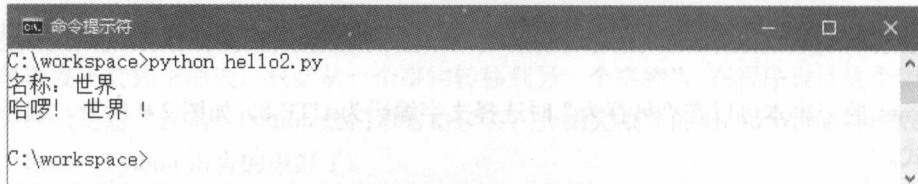


图 2-12 加上 #coding: MS936 之后，就可以正确地运行程序了

在 hello2.py 的程序代码中，input() 是个函数（Python 2.x 中使用的是 raw_input()），可用来获取用户输入的文字，调用函数时指定的文字会作为控制台中的提示信息，在用户输入文字并按【Enter】键后，input() 函数会以字符串返回用户输入的文字，在这个程序中将之赋值给变量 name，之后使用 print() 函数按序显示 '哈啰'、name 与 '!'。

为什么说是最简单的设置方式呢？将来读者也许还会看到其他的编码设置方式，例如：

```
# -*- coding: GBK -*-
```

或者是：

```
# vim: set fileencoding=GBK :
```

也许读者还会看到更多的方式，这是因为 python 解释器只要在注释中看到 coding=<encoding name> 或者 coding: <encoding name> 出现就可以了。因此，就算读者在第一行编写了 #orzcoding=MS936 或者 #XDCoding: MS936，python 解释器也都可以正确找出文字编码的设置。

提示>>>

这是为了顺应各种编辑器的特性，如果有兴趣，可以参考 PEP 0263：

www.python.org/dev/peps/pep-0263/

这个文档中有说明，python 解释器会使用下面这段正则表达式（Regular expression）来提取文字编码的设置。

```
^[\t\v]*#.*?coding[:=][\t]*([_a-zA-Z0-9]+)
```

本书第 11 章会介绍什么是正则表达式，以及在 Python 中如何使用它。

2.2 初识模块与软件包

对于初学者来说,通常只需要一个.py源码文件就可以应付各种范例程序的代码量。然而,实际的应用程序需要的代码量远比范例程序要多得多,只使用一个.py源码文件来编写,势必会造成程序代码管理上的混乱,读者必须学会根据功能将程序代码划分到不同的模块(Module)中编写,对于功能相近或彼此辅助的模块,也要知道如何使用软件包(Package)来加以管理。

模块与软件包也有一些要知道的细节,这一节将只介绍简单的入门,这样足以完成前面几个章节的学习,更详细的模块与软件包说明,将会在第5章与第14章详细介绍。

2.2.1 模块简介

有一个事实也许会令人惊讶,其实读者到这里已经编写过模块了,每个.py文件本身就是一个模块,当读者编写完一个.py文件,如果别人打算直接分享你的成果,只需要在他编写的.py文件中导入(import)就可以了。举个例子来说,若想在 hello3.py 文件中直接使用前面编写好的 hello2.py 文件,可以编写如图 2-13 所示的程序。

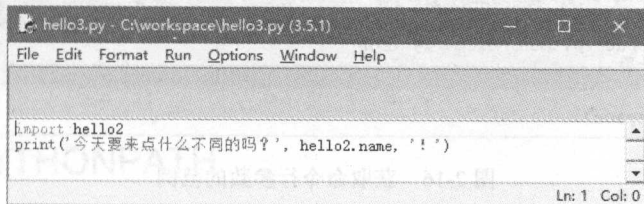


图 2-13 导入模块

每个.py文件的主文件名就是模块名称,想要导入模块时必须使用 **import** 关键词来指定模块名称。若要引用模块中定义的名称,则必须在名称前加上模块名称与一个“.”符号,例如 `hello2.name`。接下来直接执行 `hello3.py`,看看会有什么结果,如图 2-14 所示。

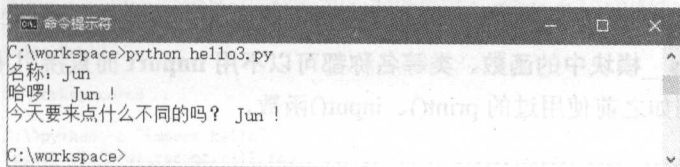


图 2-14 导入了另一模块后的程序运行结果

运行结果中前两行显示的就是 `hello2.py` 中编写的内容,即按照 **import** 的顺序导入的程序代码先被执行,接着才执行 `hello3.py` 中 **import** 之后的程序代码。

提示 >>>

此时如果查看.py文件所在的文件夹,就会发现多了一个 `__pycache__` 文件夹,当中会有.pyc文件,这是CPython将.py文件编译后的字节码文件。如果之后再次导入同一个模块,检测到源码文件没有更改过,就不会再对源码重头开始进行语义和语法解析等操作,可以直接从字节码开始解释,以加快解释执行的速度。



类似地, Python 本身提供了标准链接库, 如果需要这些链接库中的某个模块功能, 那么可以将模块导入。例如, 若想要获取命令行参数 (Command-line argument), 可以通过 `sys` 模块中的 `argv` 列表 (list), 如图 2-15 所示。

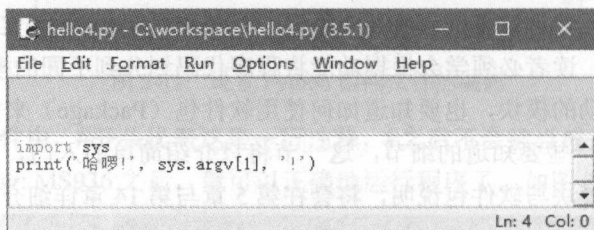


图 2-15 获取命令行参数

由于 `argv` 定义在 `sys` 模块中, 在 `import sys` 后, 就必须使用 `sys.argv` 来引用, `sys.argv` 列表中的数据引用时必须指定索引 (Index) 号码, 这个号码实际上从 0 开始, 然而 `sys.argv[0]` 保存的是源码的文件名, 就上面的例子来说, 保存的是 'hello4.py', 若要引用命令行参数, 则按序从 `sys.argv[1]` 开始引用即可。这个程序的运行结果如图 2-16 所示。

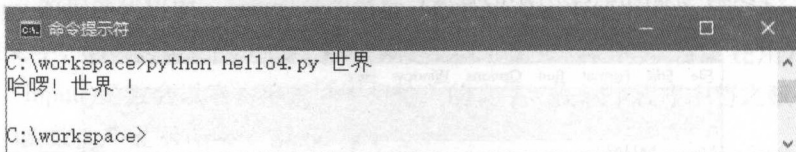


图 2-16 获取命令行参数的范例

如果有多个模块需要 `import`, 除了逐行 `import` 之外, 也可以在单独一行中使用逗号 (,) 来分隔各个模块的名称。例如:

```
import sys, email
```

使用模块来管理源码, 有利于源码的重复使用而且可以避免混乱, 然而有些函数、类等经常使用, 每次都要 `import` 就显得很麻烦, 因此这类常用的函数、类等也会被整理在一个 `__builtins__` 模块中, 在 `__builtins__` 模块中的函数、类等名称都可以不用 `import` 而直接引用, 而且不用加上模块名称作为前置, 例如之前使用过的 `print()`、`input()` 函数。

提示 >>>

想知道还有哪些函数或类吗? 可以在 REPL 中使用 `dir()` 函数查询 `__builtins__` 模块, `dir()` 函数会将可用的名称列出, 例如 `dir(__builtins__)`, 如图 2-17 所示。

```

C:\workspace>python
Python 3.5.1 |Anaconda 4.0.0 (64-bit)| (default, Feb 16 2016, 09:49:46) [MSC v.1
900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> dir (__builtins__)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '__builtins__' is not defined
>>> dir (__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'Blocki
ngIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError
', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'Conne
ctionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError
', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPoint
Error', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarni
ng', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError',
'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'Non
e', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'O
verflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupErr
or', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'Run
timeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarni

```

图 2-17 查询 `__builtins__` 模块

在官方网站文件中，也有一些 `__builtins__` 模块中函数、常数的说明文件。

docs.python.org/3.5/library/functions.html

docs.python.org/3.5/library/constants.html

2.2.2 设置 PYTHONPATH

如果已经学会了使用模块，现在有个小问题，要想引用他人编写好的模块，一定要将 `.py` 文件存放到当前的工作文件夹中吗？举个例子来说，当前的 `.py` 文件都存放在 `C:\workspace`，如果执行 `python` 指令时也是在 `C:\workspace`，基本上不会有问题，如果在其他文件夹就会出错，如图 2-18 所示。

```

C:\workspace>python -c "import hello"
Hello World

C:\workspace>cd ..

C:\>python -c "import hello"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named 'hello'

C:\>

```

图 2-18 找不到 `hello` 模块

在 2.1.1 小节中讲过，`python -c` 可以指定一段小程序来运行，因此 `python -c "import hello"` 就相当于在某个 `.py` 文件中执行了 `import hello`，因此这里用来测试是否可以找到指定模块。可以看到，在找不到指定模块时，会发生 `ImportError` 错误。

如果想将他人提供的 `.py` 文件存放到其他文件夹（例如 `lib` 文件夹）中加以管理，可以设置 `PYTHONPATH` 环境变量来解决这个问题。`python` 解释器会在环境变量设置的文件夹中寻找是否

有指定模块名称对应的.py 文件。例如，在图 2-19 中设置好环境变量，就不会再发生上述的 ImportError 错误。

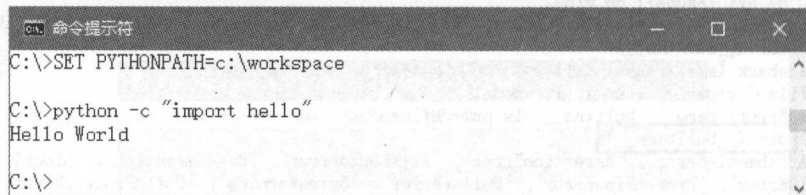


图 2-19 设置 PYTHONPATH 环境变量

在 Windows 中，可以使用 SET PYTHONPATH=路径 1;路径 2 的方式来设置 PYTHONPATH 环境变量，多个路径时中间使用分号 (;) 来分隔。实际上，python 解释器会根据 sys.path 列表中的路径来寻找模块，以当前的设置来看，sys.path 会包含以下内容，如图 2-20 所示。

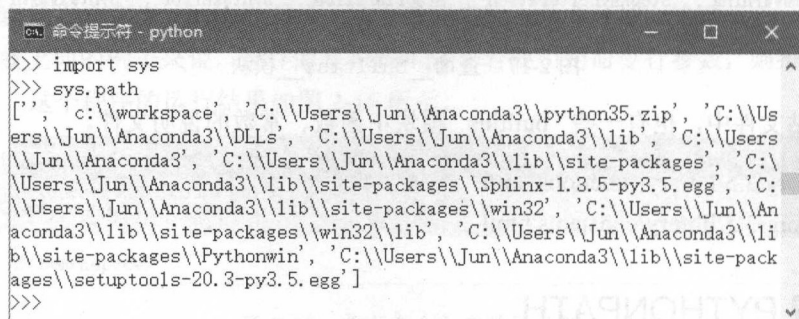


图 2-20 查询 sys.path

提示 >>>

如果 Windows 中安装了多个版本的 Python 环境，也可以按照类似方式设置 PATH 环境变量，例如 SET PATH=Python 环境路径，这样就可以切换执行不同版本的 python 解释器。

因此，如果想要动态地管理模块的寻找路径，那么也可以通过程序更改 sys.path 的内容来达到目的。例如在没有对 PYTHONPATH 设置任何信息的情况下，在进入 REPL 后，可以如图 2-21 所示进行设置。

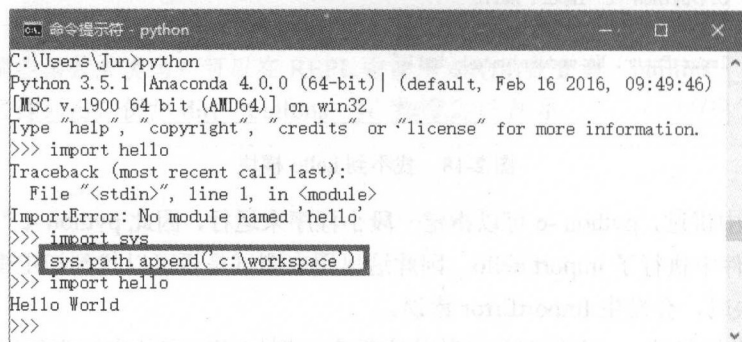


图 2-21 动态设置 sys.path

在图 2-21 中可以看到, `sys.path.append('c:\workspace')`对 `sys.path` 新增了一条路径信息, 因此之后 `import hello` 时, 就可以在 `c:\workspace` 找到对应的 `hello.py` 了。

2.2.3 使用软件包管理模块

现在读者所编写的程序代码可以分别放在各个模块之中, 在源码管理上好一些了, 但还不是很好, 就如同你会分不同文件夹来存放不同作用的文件那样, 模块也应该分门别类地存放。

举例来说, 一个应用程序中会有多个类彼此合作, 也有可能由多个团队共同分工来完成应用程序的某些功能块, 然后再组合在一起。如果应用程序是多个团队共同合作开发的, 若不分门别类地存放这些模块, 例如 A 部门写了个 `util` 模块, B 部门也写了个 `util` 模块, 当他们要将应用程序整合时, 如果都将模块存放在同一个 `lib` 目录中, 就会发生同名的 `util.py` 文件彼此覆盖的问题。

两个部门各自创建文件夹存放自己的 `util.py` 文件, 然后在 `PYTHONPATH` 中设置路径的方式是行不通的, 因为执行 `import util` 时, 只会使用通过 `PYTHONPATH` 环境变量找到的第一个 `util.py`, 真正需要的方式是能够 `import a.util` 或 `import b.util` 来导入对应的模块。

为了示范软件包的管理, 我们来创建一个新的 `hello_prj` 文件夹, 这就像新创建应用程序项目时, 必须有个项目文件夹来管理项目的相关资源。假设读者在 `hello_prj` 中新增一个 `openhome` 软件包, 那么请在 `hello_prj` 中建立一个 `openhome` 文件夹, 再在 `openhome` 文件夹中创建一个 `__init__.py` 文件, 如图 2-22 所示。

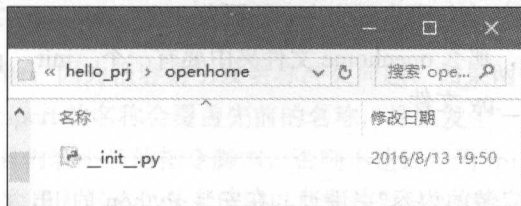


图 2-22 创建 `openhome` 软件包

注意! 文件夹中一定要有一个 `__init__.py` 文件, 该文件夹才会被视为一个软件包。在软件包的高级管理中, `__init__.py` 中其实也可以编写程序, 不过目前请保持 `__init__.py` 文件内容为空。

接着, 请将 2.1.3 小节编写的 `hello2.py` 文件复制到 `openhome` 软件包中, 然后将 2.2.1 小节编写的 `hello3.py` 文件复制到 `hello_prj` 项目文件夹中, 并修改 `hello3.py`, 如图 2-23 所示。

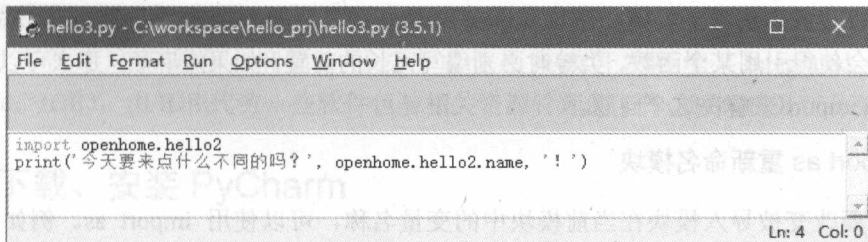
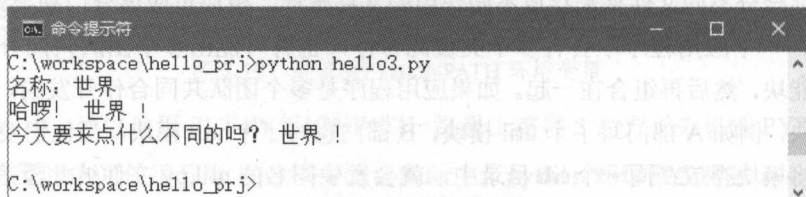


图 2-23 导入软件包中的模块

主要的修改就是 `import openhome.hello2` 与 `openhome.hello2.name`, 也就是模块名称前加上了软件包名称, 这就说明了软件包名称会成为命名空间的一部分。

当 python 解释器看到 `import openhome.hello2` 时，会寻找 `sys.path` 的路径里是否在某个文件夹中含有 `openhome` 文件夹，如果找到了，就再进一步确认其中是否有一个 `__init__.py` 文件，如果有就确认有 `openhome` 软件包了，接着查看其中是否有 `hello2.py`，如果找到了，就可以顺利完成模块的 `import`。

要执行 `hello3.py`，请在控制台（即“命令提示符”程序）中切换至 `c:\workspace\hello_prj` 文件夹，运行范例的结果如图 2-24 所示。



```

C:\workspace\hello_prj>python hello3.py
名称: 世界
哈啰! 世界!
今天要来点什么不同的吗? 世界!
C:\workspace\hello_prj>

```

图 2-24 导入软件包中模块的范例运行结果

由于软件包名称会成为命名空间的一部分，就之前 A、B 两部门的例子来说，可以分别创建 a 软件包与 b 软件包，当中存放各自的 `util.py`，当两个部分的 a、b 两个文件夹放到同一个 lib 文件夹时，就不会发生 `util.py` 文件彼此覆盖的问题。在导入模块时，可以分别 `import a.util` 与 `import b.util`，如果想引用各自模块中的名称，那么也可以使用 `a.util.some`、`b.util.other` 来区别。

如果模块数量很多，也可以创建多层次的软件包，也就是软件包中还会有软件包，在这种情况下，每个担任软件包的文件夹与子文件夹中，各要有一个 `__init__.py` 文件。举例来说，如果想要创建 `openhome.blog` 软件包，那么 `openhome` 文件夹中要有一个 `__init__.py` 文件，而 `openhome/blog` 文件夹中也要有一个 `__init__.py` 文件。

提示 >>>

还记得 1.2.3 小节“认识安装的内容”中讲过，在安装 Python 的 lib 文件夹中，包括了许多标准链接库的源码文件吗？lib 文件夹包含在 `sys.path` 中，这个文件夹中也使用了一些软件包来管理模块，其中还有个 `site-packages` 文件夹用来安装 Python 的第三程序库，这个文件夹也包含在 `sys.path` 中，通常第三程序库也会使用软件包来管理相关的模块。

2.2.4 使用 import as 与 from import

使用软件包管理解决了实体文件与 `import` 模块时命名空间的问题。然而，有时软件包名称加上模块名称会使得引用某个函数、类等时必须编写冗长的前置，如果嫌麻烦，那么可以使用 `import as` 或者 `from import` 来解决这个问题。

🔍 import as 重新命名模块

如果想要改变被导入模块在当前模块中的变量名称，可以使用 `import as`。例如可修改之前 `hello_prj` 文件夹中的 `hello3.py`，如下：

```

hello_prj2 hello3.py
import openhome.hello2 as hello

```



```
print('今天要点什么不同的吗?', hello.name, '!!')
```

在上面的范例中，`import openhome.hello2 as hello` 将 `openhome.hello2` 模块重新命名为 `hello`，接下来就可以使用 `hello` 这个名称来直接引用模块中定义的名称。

🔗 from import 直接导入名称

使用 `import as` 是将模块重新命名。然而，引用模块中定义的名称时还是得加上名称作为前置，如果仍然嫌麻烦，那么可以使用 `from import` 直接将模块中指定的名称导入。例如：

```
hello_prj3 hello.py
```

```
from sys import argv
print('哈啰!', argv[1], '!!')
```

在这个范例中，直接将 `sys` 模块中的 `argv` 名称导入到 `hello` 模块中，也就是当前的 `hello.py` 文件中，接下来就可以直接使用 `argv`，而不是 `sys.argv` 来存取命令行参数。

如果有多个名称想要直接导入到当前模块，除了逐行 `from import` 之外，还可以在单独一行中使用逗号 (,) 来分隔它们。例如：

```
from sys import argv, path
```

你可以更“偷懒”一些，用以下的 `from import` 语句来导入 `sys` 模块中全部的名称。

```
from sys import *
```

不过这个方式有点危险，因为很容易引发名称冲突问题，如果两个模块中正好都有相同的名称，那么比较后面 `from import` 的名称会覆盖先前的名称，导致发生一些意外的程序错误。因此，除非你是在编写一些简单且内容不长的指令脚本，否则不建议使用 `from xxx import *` 的方式。

`from import` 除了从模块导入名称之外，也可以从软件包导入模块，例如，如果 `openhome` 软件包下有个 `hello` 模块，就可以如下导入模块名称。

```
from openhome import hello
```

2.3 使用 IDE

在开始使用软件包管理模块后，必须建立与软件包对应的实体文件夹层次，还要自行添加 `__init__.py` 文件，这其实有点麻烦，可以考虑使用集成开发环境（Integrated Development Environment, IDE），由 IDE 代劳一些软件包与相关资源管理的工作，以便提升工作效率。

2.3.1 下载、安装 PyCharm

在 Python 的领域中，有为数不少的 IDE，然而使用哪个 IDE，必须根据开发的应用程序特性，或者基于一些团队管理等因素来决定，有时其实也是个人喜好的问题，下面介绍一些有人推荐的或使用过的 IDE。

- PyCharm (www.jetbrains.com/pycharm/)
- PyDev (www.pydev.org/)
- Komodo IDE (komodoide.com/)
- Spyder (code.google.com/archive/p/spyderlib/)
- WingIDE (wingware.com/)
- NINJA-IDE (www.ninja-ide.org/)
- Python Tools for Visual Studio (pytools.codeplex.com/)

提示 >>>

有时甚至会考虑使用一些功能强大的编辑器，加上一些插件来组装出自己专用的 IDE，在 Python 这个领域，要使用 IDE 或是编译器，也是个经常论战的话题，这当中也有一些值得思考的要点，有兴趣可以参考“IDE、编辑器的迷思”这篇文章。

openhome.cc/Gossip/Programmer/IDEorEditor.html

为了能与本书谈过的概念相衔接，作者在这里选择使用 PyCharm 进行基本介绍，它提供了社区版本，对于入门用户练习来说，完全可以胜任，读者可以直接连接到 www.jetbrains.com/pycharm/download/，单击页面右边“Community”下面的“Download”按钮，即可进行下载，如图 2-25 所示。

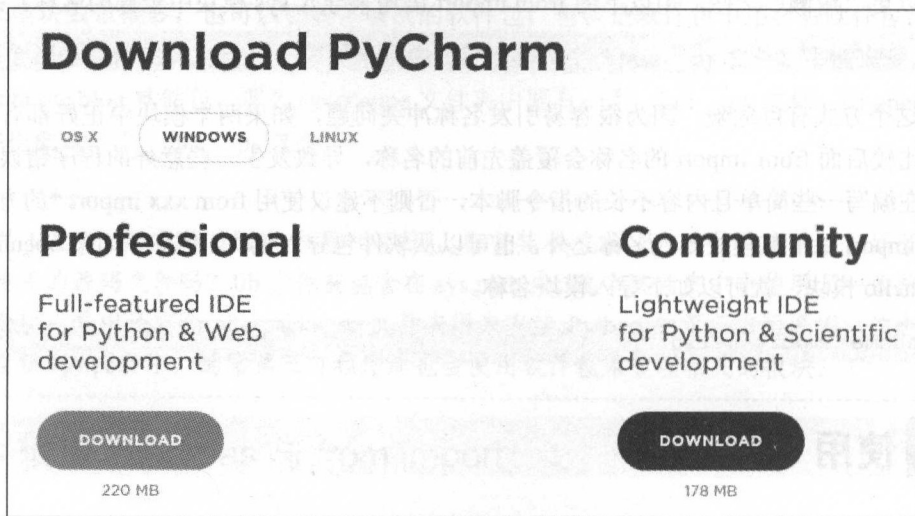


图 2-25 下载 PyCharm 社区版本

在编著本书的时候，可下载的 PyCharm Community 版本是 2016.2.1，文件为 `pycharm-community-2016.2.1.exe`。由于下载后是个 .exe 文件，读者必须如 1.2.2 小节介绍的方式“解除锁定”，并以“以管理员身份运行”进行安装。安装的默认路径是 `C:\Program Files (x86)\JetBrains\PyCharm Community Edition 2016.2.1`，基本上只需要一直单击“Next”与“Install”按钮就可以完成安装。读者可以根据自己的需要采用默认安装的目标盘和路径或者选择自己指定的安装目录和路径。

在安装完成后，应用程序菜单中会有个 JetBrains PyCharm Community Edition 2016.2.1 的图标，

单击该图标就可以启动 PyCharm，初次启动时会显示一个提示画面，询问是否导入前一版本的 PyCharm 设置，默认是不导入。由于这是初次安装，直接单击“OK”按钮即可，如图 2-26 所示。

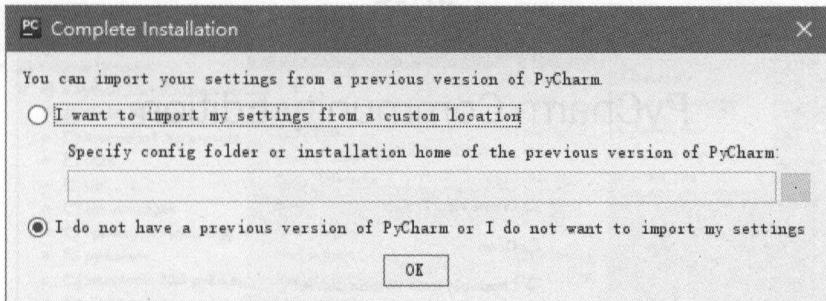


图 2-26 初次启动 PyCharm 时显示的提示画面

下一个显示画面是主题设置，如果没有特别偏好的主题，也是直接单击“OK”按钮接受默认值，如图 2-27 所示，接下来就可以准备创建新项目了。

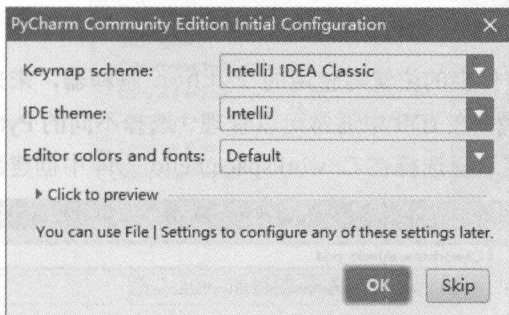


图 2-27 接受默认的主题

2.3.2 IDE 项目管理基础

IDE 基本上就是建立在当前安装的 Python 环境之上，无论使用哪个 IDE，最重要的是知道它如何与现有的 Python 环境对应，只有认清这样的对应关系，才不会沦为只知道 IDE 上一些傻瓜式的操作，却不明了各个操作背后的原理，这也是这里要介绍 IDE 的缘故。

之前在介绍软件包与模块时提到，我们会创建一个项目文件夹，在其中管理软件包、模块或其他相关资源。因此，使用 IDE 的第一步就是创建新项目，请先单击“Create New Project”，如图 2-28 所示。

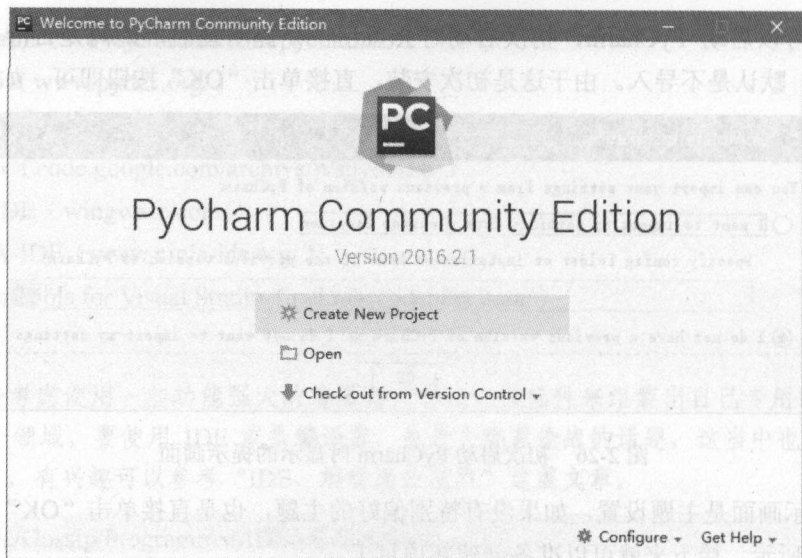


图 2-28 创建新项目

下一步是要决定项目文件夹的位置与使用的 Python 解释器，未来你的计算机中可能安装了不止一个版本的 Python 环境，在 IDE 中通常可以管理、选择不同的 Python 环境来开发程序，这也是使用 IDE 的好处之一。在这里选择在 C:\workspace\hello_prj4 中创建新项目，如图 2-29 所示。

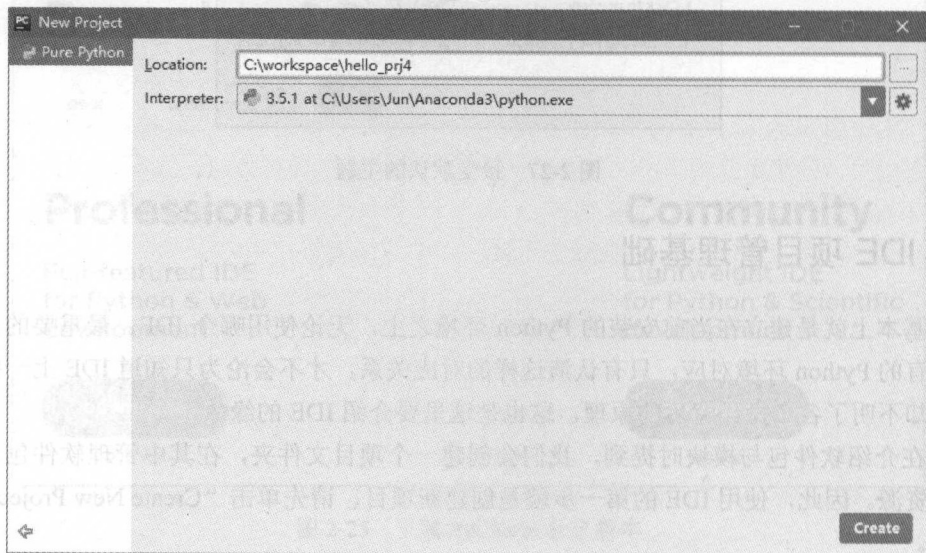


图 2-29 设置项目文件夹与解释器的版本

接着单击“Create”按钮就可以创建项目了，如图 2-30 所示。

如图 2-30 中可看到的，在“External Libraries”中可直接浏览当前使用的 python 解释器、链接库的位置等信息。读者可以试着执行“New/Python Package”来创建一个 openhome 软件包，在该软件包上执行“New/Python File”创建一个 hello.py，编写一小段程序并执行，如图 2-31 所示。

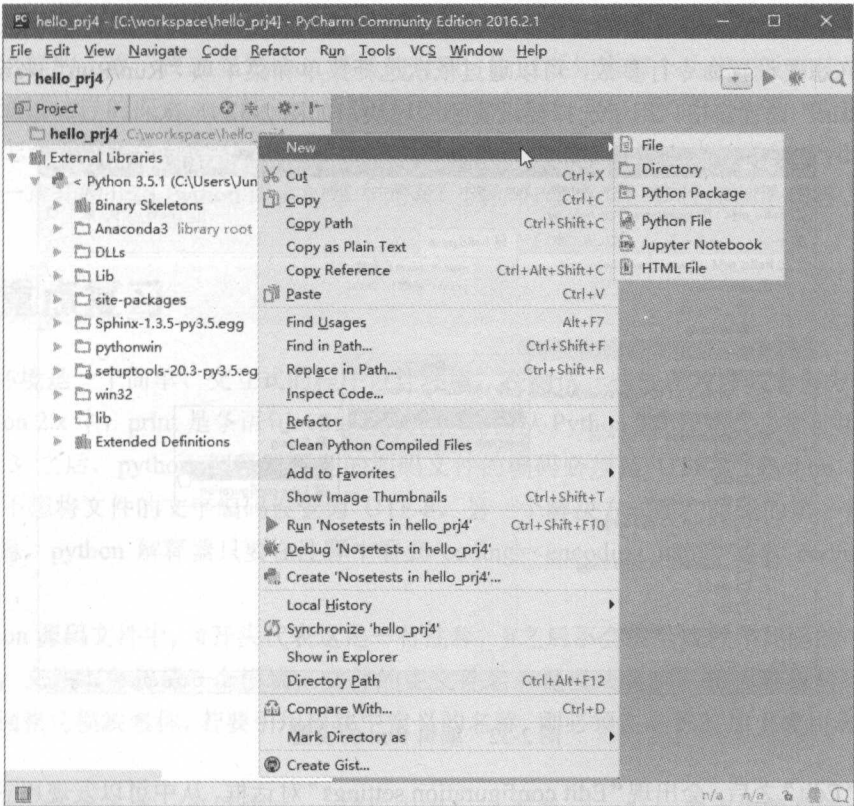


图 2-30 项目的基本结构

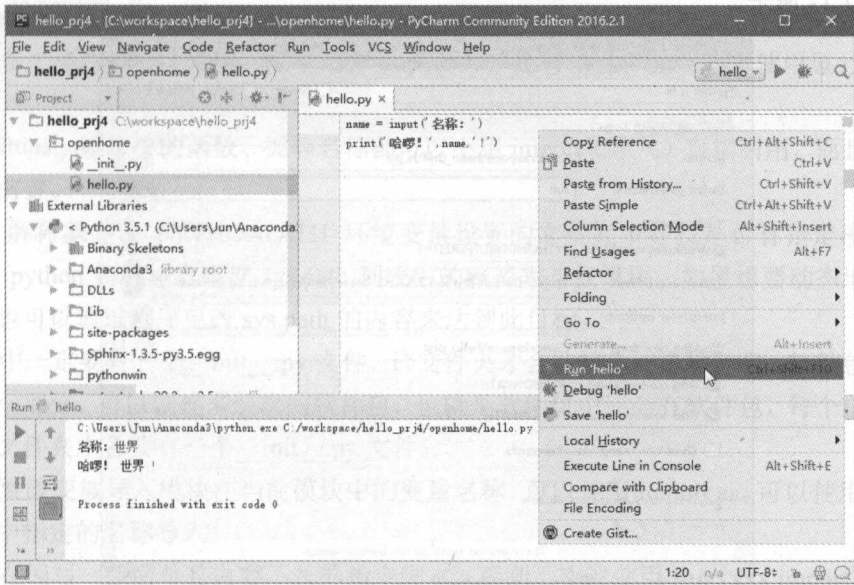


图 2-31 创建软件包、模块与运行

可以看到，在创建软件包时，IDE 会自动创建 `__init__.py`，想要执行模块，可以用鼠标右击程序，再单击弹出的快捷菜单中的“Run 'hello'”选项，其中 `hello` 会根据当前的模块名称而有所不同，

运行过的程序结果会显示在下面的窗格中，当中明确地显示了执行的过程，非常方便。

读者也许想要设置命令行参数，可以通过依次选择菜单和菜单项“Run/Run”来设置，期间会出现一个“Run”设置窗格，用于选择要设置哪个模块，如图 2-32 所示。

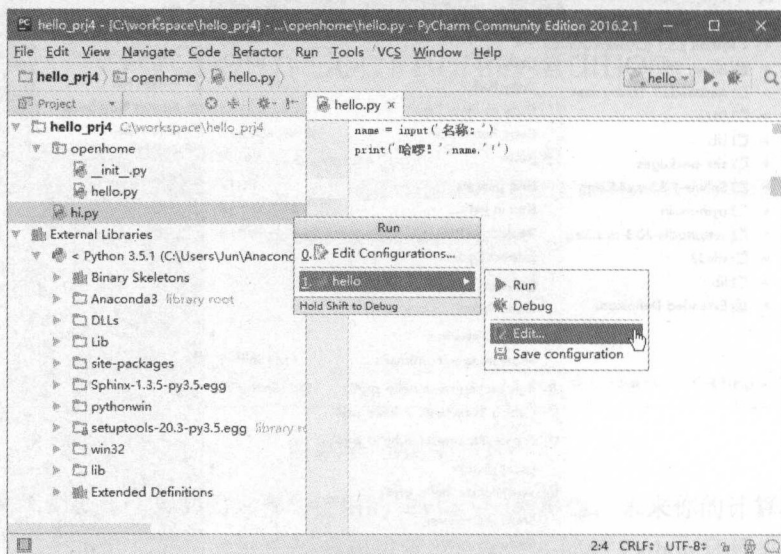


图 2-32 编辑 Run 的设置

在单击“Edit”之后，会出现“Edit configuration settings”对话框，从中可以发现用于设置 python 解释器的一些选项，像 PYTHONPATH 之类的设置，其中命令行参数可以在“Script parameters”中设置，如图 2-33 所示。

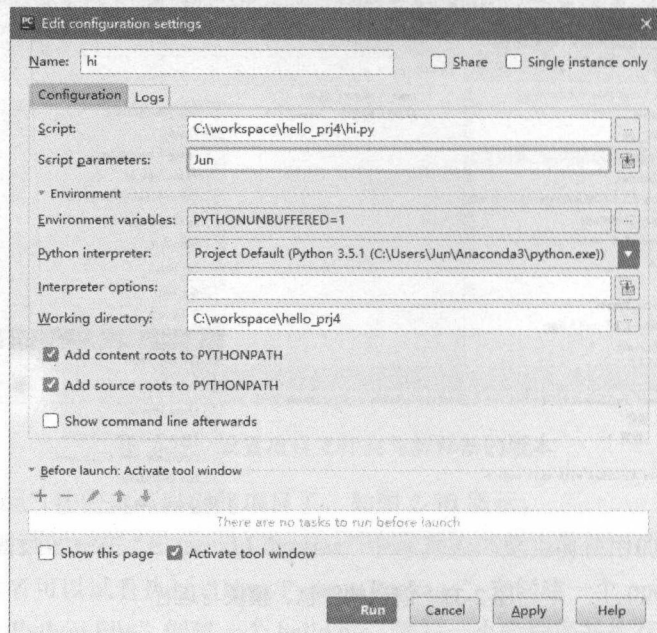


图 2-33 设置 python 解释器的相关选项

由于本书篇幅有限，不可能详细地介绍 IDE 的每个功能。不过，在开始使用一个 IDE 时，基本上也就是这样的：逐一找出与 Python 环境的对照，而且要知道哪个功能在没有使用 IDE 时是如何设置的，通过这样的探索，才能一方面获得 IDE 的方便性，另一方面又不至于被 IDE 限制。

在知道怎么编写、执行'Hello World'程序之后，接下来就要更进一步地了解 Python 程序设计语言，从下一章开始介绍 Python 语言的基本元素，例如内建类型、流程控制语句等。

2.4 重点复习

REPL 环境是一个简单、交互式的程序设计环境，在测试一些程序片段时非常方便。

在 Python 2.x 中，`print` 是条语句 (Statement)，然而从 Python 3.0 开始，必须使用 `print()` 函数。

Python 3 之后，python 解释器预期的源码文件的编码必须是 UTF-8 (Python 2.x 预期的是 ASCII)。若不想将文件的文字编码设置为 UTF-8，另一个解决方式是在源码的第一行使用注释来设置编码信息，python 解释器只要在注释中看到 `coding=<encoding name>` 或者 `coding: <encoding name>` 即可。

在 Python 源码文件中，`#` 开头代表这是一行注释，`#` 之后不会被当成程序代码的一部分。

每个 .py 文件本身就是一个模块，文件的主文件名就是模块名称，想要导入模块时必须使用 `import` 关键词指定模块名称，若要引用模块中定义的名称，则必须在名称前加上模块名称与一个“.”符号。

想要获取命令行参数，可以通过 `sys` 模块中的 `argv` 列表。`sys.argv` 列表中的数据引用时必须指定索引号码，这个号码实际上从 0 开始，`sys.argv[0]` 保存的是源码文件名，如果要引用命令行参数，就按序从 `sys.argv[1]` 开始引用。

如果有多个模块需要 `import`，除了逐行 `import` 外，还可以在单独一行中使用逗号 (,) 来分隔开模块。

在 `__builtins__` 模块中的函数、类等名称都可以不用 `import` (导入) 直接引用，而且不用加上模块名称作为前置。

python 解释器会在 `PYTHONPATH` 环境变量设置的文件夹中寻找是否有指定模块名称对应的 .py 文件。python 解释器会根据 `sys.path` 列表中的路径来寻找模块。如果想要动态地管理模块的寻找路径，也可以通过程序更改 `sys.path` 的内容来达到此目标。

文件夹中一定要有一个 `__init__.py` 文件，该文件夹才会被视为一个软件包。软件包名称会成为命名空间的一部分。可以创建多层次的软件包，也就是软件包中还会有软件包，每个担任软件包的文件夹与子文件夹中各要有一个 `__init__.py` 文件。

如果想要改变被导入模块在当前模块中的变量名称，可以使用 `import as`。可以使用 `from import` 直接将模块中指定的名称导入。

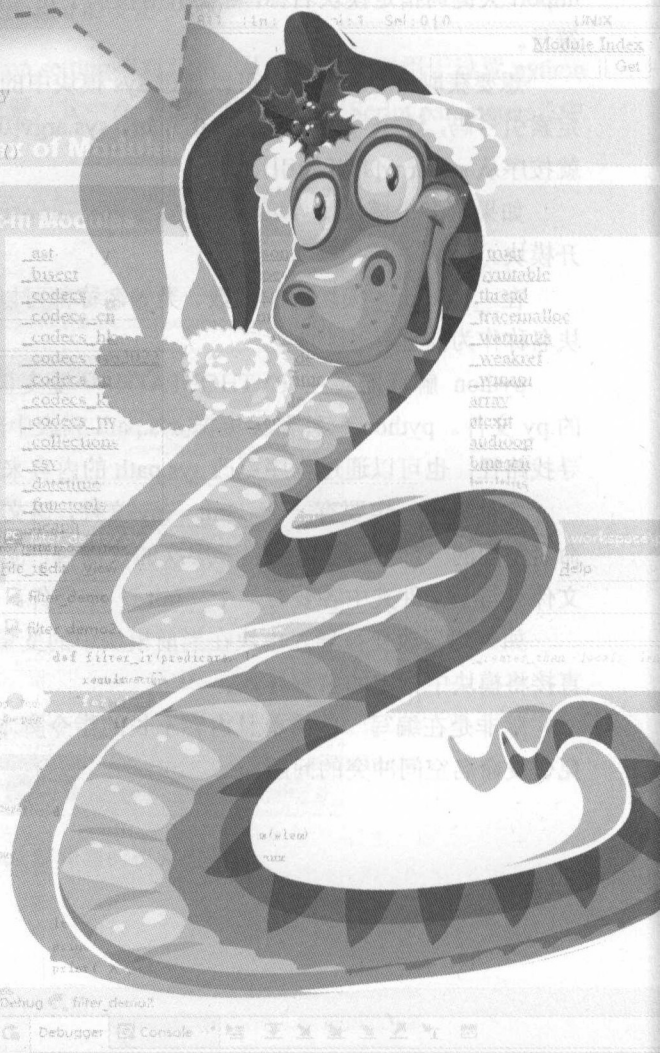
除非是在编写一些简单且内容不长的指令脚本，否则不建议使用 `from xxx import *` 的方式，以免引发命名空间冲突的问题。

第 3 章

类型与运算符

学习目标

- 认识内建类型
- 学习字符串格式化
- 了解变量与运算符
- 运用切片运算



3.1 内建类型

Python 是一个可实现多范式 (Paradigm) 的程序设计语言, 像过程式 (Procedural)、面向对象 (Object-Oriented)、函数式 (Functional) 等, 然而在正式探讨这类范式风格的实现之前, 对于语言的基础元素, 必须要有一定的认识。那么要从哪个开始呢? Pascal 之父 Niklaus E. Wirth 曾说过:

Algorithms + Data Structures = Programs

算法加数据结构就等于程序, 一门程序设计语言提供的数据类型 (Data type)、运算符 (Operator)、程序代码封装方式等会影响算法与数据结构的实现方式。因此, 在这一章会先来说明内建类型、变量、运算符等元素, 它们在 Python 语言中如何表现, 至于基本流程语句、函数、类 (Class) 等内容, 将在之后各章再分别说明。

3.1.1 数值类型

在 Python 中, 数值类型有整数、浮点数、布尔与复数, 所有的数据都是对象, 但我们可以使用直接表示法 (Literal) 来编写数值。实际上, 这一章所谓的内建类型, 是指内建在 python 解释器中, 能以直接表示法来建立实例的类型。

提示

在 Python 中, 万物皆对象! 不过, 面向对象并非 Python 的主要范式, Python 创建者 Guido van Rossum 曾经说过, 自己并非面向对象的信徒。

在《Masterminds of Programming》书中, Guido van Rossum 也谈到: “Python 支持过程式的程序设计以及 (某些程度) 面向对象。这两者没太大不同, 然而 Python 的过程式风格仍强烈受到面向对象程序设计的影响 (因为基础的数据类型都是对象)。Python 支持小部分函数式 (Functional) 程序设计, 不过它不像任何真正的函数式语言。”

整数类型

从 Python 3 之后, 整数类型为 int, 不再区分整数与长整数 (在 Python 2.x 中分别是 int 与 long 两种类型), 整数的长度不受限制 (除了硬件物理上的限制之外)。如果直接写下一个整数值, 例如 10, 默认是十进制整数。若要编写二进制数字, 则在数字前置 0b 或 0B; 若要编写八进制整数, 则在数字前置 0o 或 0O, 之后接上 1~7 的数字; 若要编写十六进制整数, 则以 0x 或 0X 开头, 之后接上 1~9 以及 A~F。例如, 下面的写法都相当于十进制整数 10。

```
>>> 10
10
>>> 0b1010
10
>>> 0o12
10
>>> 0xA
10
>>>
```

无论是十进制、二进制、八进制或十六进制整数，都是 `int` 类型的实例，想知道某个数据的类型，可以使用 `type()` 函数。例如：

```
>>> type(10)
<class 'int'>
>>> type(0b1010)
<class 'int'>
>>> type(0o12)
<class 'int'>
>>> type(0xA)
<class 'int'>
>>>
```

若想从字符串、浮点数、布尔等类型产生整数，则可以使用 `int()`，浮点数的小数会被截去，布尔值的 `True` 会返回 1，`False` 会返回 0，而使用 `oct()`、`hex()` 可以将十进制整数分别以八进制、十六进制表示的字符串形式返回。例如：

```
>>> int('10')
10
>>> int(3.14)
3
>>> int(True)
1
>>> int(False)
0
>>> oct(10)
'0o12'
>>> hex(10)
'0xa'
>>>
```

使用 `int()` 从字符串生成整数时，默认会将字符串当成十进制数来进行解析，然而也可以指定基底（即采用的进制）。例如：

```
>>> int('10', 2)
2
>>> int('10', 8)
8
>>> int('10', 16)
16
>>>
```

提示 >>>

“对象（Object）”与“实例（Instance）”这两个名词，就技术上而言略有不同，不过在本书中会将它们当成是相同的意思，就像有时会说“1 是 `int` 对象”，有时会说“1 是 `int` 的实例”。

● 浮点数类型

浮点数是 `float` 类型，可以使用 `3.14e-10` 这样的表示法，如果想将字符串解析为浮点数，就可以使用 `float()`。例如：

```
>>> type(3.14)
<class 'float'>
>>> 3.14e-10
3.14e-10
>>> float('1.414')
1.414
```

```
>>>
```

如果有个字符串是'3.14'，想要获取整数部分并解析为 int，直接使用 int() 会出现 ValueError 错误，要先使用 float() 解析为 float，接着再用 int() 获取 int 整数。例如：

```
>>> int('3.14')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '3.14'
>>> int(float('3.14'))
3
>>>
```

布尔类型

布尔类型的值只有两个，分别是 True 与 False，为 bool 类型。可以使用 bool() 将 0 转换为 False，而非 0 值转换为 True，为什么不是只有 1 能转换为 True？

实际上，bool() 可以传入任何类型，就目前为止可以先知道的是，将 None、False、0、0.0、Gj（复数）、"（空字符串）、()（空元组）、[]（空列表）、{}（空字典）等传给 bool()，都会返回 False，若这些类型的其他值传入 bool()，则都会返回 True。

提示 >>>

第 5 章会学到自定义类，在自定义类时可以定义 __bool__() 或 __len__() 方法（Method），如果这些方法返回 False 或整数 0，就会被 bool() 转换为 False。

复数

Python 支持复数的直接表示法，编写时使用 a + bj 的形式，复数是 complex 类的实例，可以直接对复数进行数值运算，例如：

```
>>> a = 3 + 2j
>>> b = 5 + 3j
>>> a + b
(8+5j)
>>> type(a)
<class 'complex'>
>>>
```

提示 >>>

虽然在数学界，复数会使用 i 来代表虚数的部分，然而在电子电气相关的工程领域，惯例却是使用 j 来表示虚部，这是因为 i 在这类工程领域，往往被用来表示电流的符号，Python 的复数表示方式显然也受到了这样的影响。

3.1.2 字符串类型

如果要在 Python 中表示字符串，可以使用 ' 或 " 来括住字符串，两者在 Python 中具有相同的作用，在 Python 3 之后的版本都是产生 str 实例，可视情况互换。例如：

```
>>> "Just'in"
"Just'in"
>>> 'Just"in'
```



```
'Just"in'
>>> 'c:\workspace'
'c:\\workspace'
>>> "c:\workspace"
'c:\\workspace'
>>>
```

基本字符串的表示

单引号或双引号的字符串表示法在 Python 中可以交替使用，就像上例中，若要在字符串中包含单引号，则使用双引号括住字符串，反之亦然，然而多数 Python 开发者的习惯是使用单引号。

在某些情况下，不用特别对\进行转义（Escape），Python 会自动将之视为\\，然而在下面这种情况下就需要了。

```
>>> print('c:\todo')
c: odo
>>> print('c:\\todo')
c:\todo
>>>
```

第一个范例中显示的结果中间有个 Tab 空格，这是因为\t在 Python 中是 Tab 的转义表示方式，其他表示方式如表 3.1 所示。

表 3.1 常用的字符串转义表示方式

符号	说明
\\	反斜线
\'	单引号，当使用"来表示字符串，又要表示单引号时使用，例如'Justin's Website'
\"	双引号，当使用""来表示字符串，又要表示双引号时使用，例如"\"text\" is a string"
\ooo	以八进制数字指定字符码点（Code point），最多三位数，例如\101'表示字符串'A'
\xhh	以十六进制数字指定字符码点，至少两位数，例如\0x41'表示字符串'A'
\uhhhh	以 16 位十六进制数值指定字符，例如\u54C8\u56C9'表示'哈啰'
\Uhhhhhhh	以 32 位十六进制数值指定字符，例如\U000054C8\U000056C9'表示'哈啰'
\0	空字符，请别与空字符串搞混，\0'相当于'\x00'
\n	换行
\r	回车
\t	Tab

因此，如果想要以字符串表示\t 这类字符，就必须编写\\t，是有些不方便，这时可以使用原始字符串（Raw String）来表示，只要在字符串前加上 r 即可。例如：

```
>>> print('\\t')
\t
>>> print(r'\t')
\t
>>> r'\t'
'\\t'
>>> print(r'c:\todo')
c:\todo
>>>
```

使用"""或'''表示字符串时，不可以换行。如果字符串内容必须跨越数行，可以使用三重引号，在三重引号之间输入的任何内容，在最后的字符串会照单全收，包括换行、缩排等。例如：


```
>>> '''Justin is caterpillar!
...   caterpillar is Justin!'''
'Justin is caterpillar!\n  caterpillar is Justin!'
>>> print('Justin is caterpillar!
...   caterpillar is Justin!')
Justin is caterpillar!
  caterpillar is Justin!
>>>
```

在 REPL 中，若程序代码编写过程中有换行，则会使用“...”来提示，因此上面的范例中“...”并不是跨行字符串时要输入的部分，使用 REPL 是为了方便看到多行字符串换行时的“\n”表示。

可以使用 `str()` 类将数值转换为字符串。若想知道某个字符的码点（即编码），则可以使用 `ord()` 函数，使用 `chr()` 则可以将指定码点转换为字符。

```
>>> str(3.14)
'3.14'
>>> ord('哈')
21704
>>> chr(21704)
'哈'
>>>
```

格式化字符串

至今为止，为了在控制台显示输出结果，只使用过 `print()` 函数，当需要在一行中显示多个字符串时，可在调用 `print()` 函数时，以逗号“,”来分隔多个字符串，例如：

```
name = 'Justin'
print('Hello ', name)
```

`print()` 函数的显示默认会换行，`print()` 有个 `end` 参数，在指定的字符串显示之后，`end` 参数指定的字符串才会输出，因此如果不想换行，那么只要将 `end` 指定为空字符串“”即可。例如，下面这个程序片段的输出结果与上面的程序片段是相同的。

```
name = 'Justin'
print('Hello ', end = '')
print(name)
```

当指定多个字符串给 `print()` 函数时，默认的分隔符是一个空格符，如果想要指定其他字符，那么可以指定 `sep` 参数。例如：

```
name = 'Justin'
print('Hello', name, sep = ', ') # 显示 Hello, Justin
```

除了指定 `end` 与 `sep` 参数之外，读者也许还有其他的显示格式需求，在其他程序设计语言中，也许会有 `printf()` 之类的函数来完成这类的显示格式需求，不过 Python 中并没有这类函数。相对地，我们必须直接对字符串进行格式化，接着将格式化后的字符串交给 `print()` 函数。

虽然本节的内容只是有关类型的基本认识，然而格式化字符串这件事，也许比读者想象的还要重要且常用，因此要在这里先做个介绍。目前的 Python 3 支持两种格式化方式，一种旧式（从 Python 2 就存在），一种新式（从 Python 2.6、2.7 开始支持），目前这两种方式都很常见，因此本节都会加以介绍。

旧的字符串格式化是使用 `string % data` 或 `string % (data1, data2, ...)` 的方式，直接来看个范例会

比较清楚。

```
>>> '哈啰! %s!' % '世界'
'哈啰! 世界!'
>>> '你目前的存款只剩 %f 元' % 1000
'你目前的存款只剩 1000.000000 元'
>>> '%d 除以 %d 是 %f' % (10, 3, 10 / 3)
'10 除以 3 是 3.333333'
>>> '%d 除以 %d 是 %.2f' % (10, 3, 10 / 3)
'10 除以 3 是 3.33'
>>> '%5d 除以 %5d 是 %.2f' % (10, 3, 10 / 3)
' 10 除以      3 是 3.33'
>>> '%-5d 除以 %-5d 是 %.2f' % (10, 3, 10 / 3)
'10   除以 3      是 3.33'
>>> '%-5d 除以 %-5d 是 %10.2f' % (10, 3, 10 / 3)
'10   除以 3      是      3.33'
>>>
```

也就是%前面的字符串中会有一些占位用的控制符号，像%s、%d、%f 分别表示其中会有一个字符串、整数、浮点数，%之后的()中要按序摆放实际的值，如果只有一个控制符号需要取代，那么可以不使用()。常见的控制符号如表 3.2 所示。

表 3.2 常用格式化的控制符号

符号	说明
%%	因为%符号已经被用来作为控制符号的前置，所以规定使用%%才能在字符串中表示%
%d	十进制整数
%f	十进制浮点数
%g	十进制整数或浮点数
%e,%E	科学计数法的浮点数格式化，%e 表示输出以小写表示，如 2.13 e+12，%E 表示以大写表示
%o	八进制整数
%x,%X	以十六进制整数的格式化，%x 表示字母输出以小写表示，%X 则以大写表示
%s	字符串格式符号
%r	以 repr()函数取得的结果输出字符串，本章稍后会谈到 repr()

在之前的范例中也可以看到，%之后还可以指定最小数字宽度与小数点后保留的位数。例如%5d 表示最少保留五个数字宽度给整数使用，如果整数不足五个位数，就使用空格表示，若加上减号(-)则表示靠左对齐，%.2f 表示小数点后保留两个位数。如果不想固定“写死”数字宽度或小数点，那么也可以用%*.*来表示。例如：

```
>>> n1 = 10
>>> n2 = 3
>>> '%-5d 除以 %-5d 是 %10.2f' % (n1, n2, n1 / n2)
'10   除以 3      是      3.33'
>>> '%*d 除以 %*d 是 %*.*f' % (-5, n1, -5, n2, 10, 2, n1 / n2)
'10   除以 3      是      3.33'
>>>
```

可以看到，旧的格式化方式在一些复杂的格式化需求下，可读性并不好，如果使用 Python 3 之后的版本（或者 Python 2.6、2.7），建议使用新的格式化方式。直接来看几个例子：

```
>>> '{} 除以 {} 是 {}'.format(10, 3, 10 / 3)
```

```
'10 除以 3 是 3.3333333333333335'
>>> '{2} 除以 {1} 是 {0}{}'.format(10 / 3, 3, 10)
'10 除以 3 是 3.3333333333333335'
>>> '{n1} 除以 {n2} 是 {result}{}'.format(result = 10 / 3, n1 = 10, n2 = 3)
'10 除以 3 是 3.3333333333333335'
>>>
```

可以看到，占位符号的部分使用{}，若当中没有数字或名称，format()方法中就要按序指定对应的数值，如果{}中有数字，例如{1}，就表示使用format()方法中的第二个参数，这是因为索引值从0开始。如果{}中指定了名称，例如{n1}，就表示使用format()中的具名参数n1对应的值，在这种情况下，不用在意n1、n2、result在format()中的指定顺序。

无论是{0}或者是{n}这样的方式，都可以像旧的格式化那样指定类型，也可以指定数字宽度与小数点后的位数，例如：

```
>>> '{0:d} 除以 {1:d} 是 {2:f}'.format(10, 3, 10 / 3)
'10 除以 3 是 3.333333'
>>> '{0:5d} 除以 {1:5d} 是 {2:10.2f}'.format(10, 3, 10 / 3)
' 10 除以      3 是      3.33'
>>> '{n1:5d} 除以 {n2:5d} 是 {r:.2f}'.format(n1 = 10, n2 = 3, r = 10 / 3)
' 10 除以      3 是 3.33'
>>> '{n1:<5d} 除以 {n2:<5d} 是 {r:.2f}'.format(n1 = 10, n2 = 3, r = 10 / 3)
'10   除以 3      是 3.33'
>>> '{n1:>5d} 除以 {n2:>5d} 是 {r:.2f}'.format(n1 = 10, n2 = 3, r = 10 / 3)
' 10 除以      3 是 3.33'
>>> '{n1:*^5d} 除以 {n2:*^5d} 是 {r:.2f}'.format(n1 = 10, n2 = 3, r = 10 / 3)
'*10** 除以 !!3!! 是 3.33'
>>>
```

在上面的范例中，“<”用来指定向左对齐，“>”用来指定向右对齐，如果没有指定，默认是向右对齐；如果在数字位数不足指定的数字宽度时补上指定的字符，那么可以在“^”前面指定。

format()方法甚至可以进行简单的运算，像使用索引获取列表元素的值、使用键（Key）名称获取字典中对应的值或者引用模块中的名称，例如：

```
>>> names = ['Justin', 'Monica', 'Irene']
>>> 'All Names: {n[0]}, {n[1]}, {n[2]}'.format(n = names)
'All Names: Justin, Monica, Irene'
>>> passwords = {'Justin': 123456, 'Monica': 654321}
>>> 'The password of Justin is {passwds[Justin]}'.format(passwds = passwords)
'The password of Justin is 123456'
>>> import sys
>>> 'My platform is {pc.platform}'.format(pc = sys)
'My platform is win32'
>>>
```

关于列表（list）与字典（dict），稍后就会说明。若只是要格式化单个数值，则可以使用format()函数。例如：

```
>>> format(3.14159, '.2f')
'3.14'
>>>
```

str 与 bytes

在1.1.1小节中就谈到了，Python 3最引人注目的是支持Unicode，将str/unicode进行了整合，并明确地提供了另一个bytes类型，解决了许多人处理字符编码的问题，在本节就来解释一下str、

unicode 与 bytes 之间的关系。

先来谈一下 len() 函数，它可以用来获取一个字符串的长度，那么你觉得 len('哈') 会得到的数字是多少？应该是 1？如果是 Python 3 之后的版本，这个答案是正确的。

从 Python 3 之后，每个字符串都包含了 Unicode 字符，每个字符串都是 str 类型。如果想将字符串转为指定的编码，可以使用 encode() 方法来指定编码，结果是一个 bytes 实例，当中包含了字节数据。如果有一个 bytes 实例，也可以使用 decode() 方法指定该字节数据代表的编码，即将 bytes 解码为 str 实例。

```
>>> text = '哈'
>>> len(text)
1
>>> text.encode('UTF-8')
b'\xe5\x93\x88'
>>> text.encode('GBK')
b'\xb9\xfe'
>>> gbk_impl = text.encode('GBK')
>>> type(gbk_impl)
<class 'bytes'>
>>> gbk_impl.decode('GBK')
'哈'
>>>
```

当使用 UTF-8 编码来表示一个汉字时，需要用到三个字节。因此在上面，'哈'使用了三个字节 b'\xe5\x93\x88'，而 GBK 编码表示一个汉字时，会用到两个字节，也就是 b'\xb9\xfe'。

提示 >>>

搞不清楚谁可以用 encode() 而谁可以用 decode() 吗？'\xe5\x93\x88' 这样的信息，人们不容易理解，将可理解的东西变为不容易理解的信息就是编码 (encode())，反之将不易理解的信息变为可理解的东西就是解码 (decode())。

在 REPL 的显示结果中也可以看到，可以在字符串前加上个 b 来创建一个 bytes，这是从 Python 3.3 之后开始支持的语法。同样，也可以在字符串前加上一个 u，结果就是一个 str。

在 Python 3 之后，字符串默认就是 str，因此不用特别加上 u，这个语法是为了增加与 Python 2 的兼容性而增加的。在 Python 2 中，如果有一个 u'哈啰' 字符串，就会创建一个 unicode，而 len(u'哈啰') 的结果是 2。然而，如果只是编写'哈啰' 字符串（注意字符串前面没有 u），就会创建一个 str，然而 len('哈啰') 的结果，令人意外地，就要看源码文件的文字编码而定了，如果是 UTF-8 编码，那么结果会是 6，如果是 GBK 编码，那么结果会是 4。

这是因为在 Python 2 中，str 实际上代表着字符串编码实现的字节序列 (Byte sequence)，而 len() 函数返回的就是字节序列的长度，而不是字符长度，为了支持 Unicode，才有了 u'哈啰' 这样的语法。相对地，在 Python 2 中，一个 str 可以使用 decode() 指定编码，返回一个 unicode，而一个 unicode 可以使用 encode() 指定编码，返回一个 str。

从 Python 3 之后，想要获取字符串中某个位置的字符时，可以使用索引（或称为下标），索引从 0 开始。想要测试某字符是否在字符串中，可以使用 in。例如：

```
>>> text = '哈啰'
>>> text[0]
```



```
'哈'
>>> text[1]
'呀'
>>>
>>> '哈' in text
True
>>>
```

不过在 Python 2 中要注意，使用索引获取字符串中的值，其实是存取某个位置的字节。虽然可以使用索引对字符串取值，然而无论是 Python 2 还是 Python 3，字符串都是不可变动的 (Immutable)，字符串一旦创建，就无法修改它的内容。

3.1.3 群集类型

在编写程序的过程中，经常需要收集一些数据以备后续处理，要满足不同的需求，就需要不同的数据结构来收集数据。例如可能会需要有序的列表 (list)、不重复的集合 (set)、键值对应的字典 (dict) 等。在其他程序设计语言中，要创建这类数据结构，可能使用标准链接库来提供，然而在 Python 中，要创建这类常用的数据结构，语言本身就支持，这也是 Python 易于使用的一大原因。

接下来将会简单介绍 Python 常用的群集类型，然而这些群集类型功能强大。有关更多强大的 API 使用，在之后的章节还会有详细说明。

列表 (list)

列表的类型是 list，特性为有序、具备索引，内容与长度可以变动。要创建列表，可以使用 [] 直接表示法，列表中每个元素使用逗号 (,) 分隔开。例如：

```
>>> numbers = [1, 2, 3]
>>> numbers
[1, 2, 3]
>>> numbers.append(4)
>>> numbers
[1, 2, 3, 4]
>>> numbers[0]
1
>>> numbers[1]
2
>>> numbers[3] = 0
>>> numbers
[1, 2, 3, 0]
>>> numbers.remove(0)
>>> numbers
[1, 2, 3]
>>> del numbers[0]
>>> numbers
[2, 3]
>>> 2 in numbers
True
>>>
```

可以使用 [] 创建长度为 0 的列表，可以对列表使用 append()、pop()、remove()、reverse()、sort() 等方法。若要附加多个元素，则可以使用 extend() 方法，例如 numbers.extend([10, 20, 30])。想要知道列表中是否含有某元素，可以使用 in；想知道长度，可以使用 len()。列表中的元素通常是同质

中不会有任何显示), `get()`也可以指定默认值, 在指定的键不存在时就返回默认值。

如果要获取字典中的每一对键与值, 那么可以使用 `items()`方法, 以 `dict_items` 对象通过(Tuple) 获取每一对键与值。只要获取键, 就可以使用 `keys()`方法, 返回 `dict_keys` 对象可以逐一获取每个键。如果要获取值, 那么可以使用 `values()`方法来返回 `dict_values` 对象, 可以逐一获取每个值 `dict_items`、`keys`、`dict_values`。因为都是可迭代对象, 因此可以传给 `list()`, 创建一个列表来包含其中全部的值。

```
>>> passwords = {'Justin': 123456, 'caterpillar' : 933933}
>>> list(passwords.items())
[('caterpillar', 933933), ('Justin', 123456)]
>>> list(passwords.keys())
['caterpillar', 'Justin']
>>> list(passwords.values())
[933933, 123456]
>>>
```

提示

在 Python 2 中, 字典的 `items()`、`keys()`、`values()`会返回列表, 然而若字典中有很多成对的键与值, 与创建一个够长的列表来存储这些元素相比, Python 3 的做法更加经济, 因为 Python 3 的 `dict_items`、`dict_keys`、`dict_values` 返回后, 还没有实际去获取字典中的对应键与值, 只有在真正需要下一个元素时, 才会进行相关的运算, 这样的特性称之为惰性求值(Lazy evaluation, 也称为延迟求值)。

除了使用直接表示方式之外, 也可以使用 `dict()`来创建字典。例如:

```
>>> passwords = dict(justin = 123456, momor = 670723, hamimi = 970221)
>>> passwords
{'hamimi': 970221, 'justin': 123456, 'momor': 670723}
>>> passwords = dict([('justin', 123456), ('momor', 670723), ('hamimi', 970221)])
>>> passwords
{'hamimi': 970221, 'justin': 123456, 'momor': 670723}
>>> dict.fromkeys(['Justin', 'momor'], 'NEED_TO_CHANGE')
{'Justin': 'NEED_TO_CHANGE', 'momor': 'NEED_TO_CHANGE'}
>>>
```

元组 (tuple)

元组在许多地方都很像列表, 都是有序的结构, 可以使用[]指定索引来获取元素。不过, 元组创建之后, 就不能变动了。创建元组其实很简单, 只要在某个值后面加上一个逗号(,)即可。例如:

```
>>> 10,
(10,)
>>> 10, 20, 30,
(10, 20, 30)
>>> acct = 1, 'Justin', True
>>> acct
(1, 'Justin', True)
>>> type(acct)
<class 'tuple'>
>>>
```

创建元组时, 最后一个逗号可以省略。可以看到, 元组的类型是 `tuple`。虽然只要在值之后加上逗号就可以了, 不过通常会加上()让人一眼就看出这是个元组, 例如(1, 2, 3)、(1, 'Justin', True)。

不过要注意，只包含一个元素的元组，不能写成(elem)这样的方式，而是要写成“elem,”或者(elem,)，如果要创建没有任何元素的元组，倒是可以只写()。

提示>>>

事实上，之前在创建列表、集合或字典时，也都是省略了最后一个元素之后的逗号，日后有机会在一些程序代码中看到最后有个逗号，就不要太讶异了。

元组可以做什么用呢？有时会想要返回一组相关的值，又不想特地自定义一个类型，就会使用元组，像 (1, 'Justin', True) 也许就代表了从数据库中临时查出来的一笔数据。有时希望某个函数不要修改传入的数据，因为元组无法更改，这时就可以将数据放在元组中传入，万一函数的实现者试图修改数据，那么执行时就会出错，我们就会知道有人试图“违规”了。另外，元组占用的内存空间比较小。

可以将元组中的元素拆解 (Unpack)，逐一分配给每个变量，例如：

```
>>> data = (1, 'Justin', True)
>>> id, name, verified = data
>>> id
1
>>> name
'Justin'
>>> verified
True
>>>
```

记得吗？元组的()括号可以省略，虽然在多数情况下，可以表示是个元组，但是以下的情况省略括号，却是 Python 中最常被拿来津津乐道的特色。

```
>>> x = 10
>>> y = 20
>>> x, y = y, x
>>> x
20
>>> y
10
>>>
```

这是个置换 (Swap) 变量值的操作，在其他程序设计语言中，通常需要一个暂存变量，而在 Python 中只要一行就可以完成。

其实，无论是 Python 2 还是 Python 3，拆解元素赋值给变量的特性，在列表、集合等对象上也可以使用。例如 `x, y, z = [1, 2, 3]` 的结果，`x` 会是 1、`y` 会是 2 而 `z` 会是 3。不过，Python 3 之后，还进一步扩展了这个功能，称为 Extended Iterable Unpacking¹ (扩展可迭代拆解)，以下的功能在 Python 2 无法执行。

```
>>> a, *b = (1, 2, 3, 4, 5)
>>> a
1
>>> b
[2, 3, 4, 5]
```

¹ PEP 3132 -- Extended Iterable Unpacking: www.python.org/dev/peps/pep-3132/


```
>>> a, *b, c = [1, 2, 3, 4, 5]
>>> a
1
>>> b
[2, 3, 4]
>>> c
5
>>>
```

在某个变量上指定星号(*)，其他变量被分配了单一的值之后，剩余的元素就会以列表赋值给标上了星号的变量。

提示>>>

为什么要认识这么多类型？实际上，计算机里一切都是二进制位(bit，即比特)，对于一组有用的位数据，定义为一个数据类型就可以用具体的概念来操作这一组二进制位，而不用直接面对0101的运算。

3.2 变量与运算符

在前一节中，我们使用过变量，也在介绍Python的内建类型时对它们执行了一些基本操作，本节来进一步认识变量，并集中探讨Python的运算符在内建类型上会有什么样的表现，这也是为了突显一个事实，未来若有必要，我们也能自定义运算符的行为。

3.2.1 变量

在前一节介绍各个内建类型时多半是直接写下一个值，实际在编写程序时，这么编写是不行的。

```
print('圆半径: ', 10)
print('圆周长: ', 2 * 3.14 * 10)
print('圆面积: ', 3.14 * 10 * 10)
```

半径10同时出现在多个位置，圆周率3.14也是，如果将来要修改半径值呢？或者是要使用更精确一些的圆周率，像是3.14159呢？我们将不得不修改多个位置！

这时如果能要求Python有个位置可以暂存10与3.14这些值，每次都取出这个暂存位置的值来计算，将来只需修改暂存位置的值，那么所有计算就会取用修改后的值来运算，这样岂不是更加方便。

```
radius = 10
PI = 3.14
print('圆半径: ', radius)
print('圆周长: ', 2 * PI * radius)
print('圆面积: ', PI * radius * radius)
```

这个暂存位置在程序设计语言中就被称为**变量(Variable)**，作用之一就是修改值很方便。作用之二就如你所见，程序代码中清楚多了，原来10这个值是半径(radius)，3.14是圆周率(PI)，

而不是幻数 (Magic number¹)，而公式的部分，像 $2 * \text{PI} * \text{radius}$ ，比 $2 * 3.14 * 10$ 更有实际意义。

根据类型的信息记录在变量中或者运行时刻的对象中，程序设计语言可以区分为静态类型 (Statically-typed)、动态类型 (Dynamically-typed) 两大类。

Python 属于动态类型的程序设计语言，变量本身并没有类型信息，因此到现在可以看到，创建变量都没有声明类型，只要命名变量并使用赋值运算 “=” 把一个值赋给它，就创建了一个变量。在创建变量之前就尝试存取某个变量，会发生 `NameError` 错误，即表示变量未定义的错误。例如：

```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

在 Python 中，变量始终是个引用到实际对象的名称，赋值运算只是改变了变量的引用对象。

例如：

```
>>> x = 1.0
>>> y = x
>>> print(id(x), id(y))
25711904 25711904
>>> y = 2.0
>>> print(id(x), id(y))
25711904 25711872
>>>
```

在上面的范例中，`x` 一开始引用到浮点数对象 `1.0`，而后将 `x` 引用的对象赋值给 `y`，可以使用 `id()` 函数来获取变量引用的对象的内存地址，可以看到一开始 `x` 与 `y` 都引用同一对象，之后 `y` 引用到 `2.0`，`x` 与 `y` 就引用了不同的对象。下面这个例子也显示，变量在 Python 中只是一个引用。

```
>>> x = 1
>>> id(x)
1795962640
>>> x = x + 1
>>> id(x)
1795962656
>>>
```

一开始 `x` 引用到整数对象 `1`，在 `+` 运算后，创建了新的整数对象 `2`，而后赋值给 `x`，所以 `x` 引用至新对象。由于变量在 Python 中只是个引用至对象的名称，对于可变对象才会有以下的操作结果。

```
>>> x = [1, 2, 3]
>>> y = x
>>> x[0] = 10
>>> y
[10, 2, 3]
>>>
```

在赋值 `y = x` 时，`x` 与 `y` 就引用了同一个对象，因此将 `x[0]` 修改为 `10`，通过 `y` 就会看到修改的元素。除了使用 `id()` 查看变量引用的对象的地址值来确认两个变量是否引用同一对象之外，还可以使用 `is` 或 `is not` 运算。例如：

¹ Magic number: [en.wikipedia.org/wiki/Magic_number_\(programming\)](https://en.wikipedia.org/wiki/Magic_number_(programming))

```
>>> list1 = [1, 2, 3]
>>> list2 = [1, 2, 3]
>>> list1 == list2
True
>>> list1 is list2
False
>>>
```

稍后在介绍关系运算符时会看到，`==`运用在列表上时，可以逐一比较两个列表中的元素是否全部相等，因此在上例，`list1 == list2` 的结果是 `True`。然而，`list1` 与 `list2` 两个引用了不同的对象，因此 `list1 is list2` 的结果是 `False`。

变量本身没有类型，同一个变量可以前后赋予不同的数据类型，若想通过变量操作对象的某个方法，只要确认该对象上确实有该方法即可。例如下面 `x` 前后分别赋值了列表与元组，然而在两个对象上都有可查询元素索引位置的 `index()` 方法，因此并不会出现错误。

```
>>> x = [1, 2, 3]
>>> x.index(2)
1
>>> x = (10, 20, 30)
>>> x.index(20)
1
>>>
```

这是动态类型的程序设计语言界流行的鸭子类型 (**Duck typing**¹): “如果它走路像个鸭子，游像个鸭子，叫声像个鸭子，那它就是鸭子。”

如果想要知道一个对象有几个名称引用到它，可以使用 `sys.getrefcount()` 函数。例如：

```
>>> import sys
>>> x = [1, 2, 3]
>>> y = x
>>> z = x
>>> sys.getrefcount(x)
4
>>>
```

如果有个变量不再需要，可以使用 `del` 来删除它。例如：

```
>>> x = 10
>>> x
10
>>> del x
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

3.2.2 加减乘除运算

学习程序设计语言，加减乘除应该是很基本的，不就是使用 `+`、`-`、`*`、`/` 运算符吗？许多程序设计方面的书在这个部分的介绍也很简短。不过，实际上并不是那么简单，毕竟还是在跟计算机打交

¹ Duck typing: en.wikipedia.org/wiki/Duck_typing

道,目前还没有一个程序设计语言可以高级到完全忽略计算机的物理性质,因此有些细节还是要注意一下。

应用于数值类型

首先来看看+、-、*、/应用在数值类型时,有哪些要注意的地方。 $1 + 1$ 、 $1 - 0.1$ 对你来说都不成问题,结果分别是2、0.9,那么你认为 $0.1 + 0.1 + 0.1$ 、 $1.0 - 0.8$ 会是多少?前者不是0.3,后者也不是0.2。

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>> 1.0 - 0.8
0.19999999999999996
>>>
```

诧异吗?开发人员基本上都要知道IEEE 754浮点数算术标准¹,Python也遵守此标准,这个算术标准不使用小数点,而是使用分数和指数来表示小数,例如0.5会以 $1/2$ 来表示,0.75会以 $1/2 + 1/4$ 来表示,0.875会以 $1/2 + 1/4 + 1/8$ 来表示。然而,有些小数无法使用有限的分数来表示,像0.1,会是 $1/16 + 1/32 + 1/256 + 1/512 + 1/4096 + 1/8192 + \dots$ 没有止境,因此造成了浮点数的误差。

那么有这个误差会发生什么?如果对小数点的精度要求很高,就要小心这个问题,像最基本的 $0.1 + 0.1 + 0.1 = 0.3$,结果会错误,如果程序代码中有这类的判断,那么就会因为误差而使得程序的行为不会以你想象的方式进行。

提示>>>

在百度或者谷歌搜索中输入“算钱用浮点”,你会看到什么搜索建议呢?试试看吧!

如果需要处理小数,而且需要精确的结果,那么可以使用`decimal.Decimal`类。例如:

operator decimal_demo.py

```
import sys
import decimal

n1 = float(sys.argv[1])
n2 = float(sys.argv[2])
d1 = decimal.Decimal(sys.argv[1])
d2 = decimal.Decimal(sys.argv[2])

print('# 不使用 decimal')
print('{0} + {1} = {2}'.format(n1, n2, n1 + n2))
print('{0} - {1} = {2}'.format(n1, n2, n1 - n2))
print('{0} * {1} = {2}'.format(n1, n2, n1 * n2))
print('{0} / {1} = {2}'.format(n1, n2, n1 / n2))

print('\n# 使用 decimal')
print('{0} + {1} = {2}'.format(d1, d2, d1 + d2))
print('{0} - {1} = {2}'.format(d1, d2, d1 - d2))
print('{0} * {1} = {2}'.format(d1, d2, d1 * d2))
print('{0} / {1} = {2}'.format(d1, d2, d1 / d2))
```

¹ IEEE Standard for Floating-Point Arithmetic: en.wikipedia.org/wiki/IEEE_floating_point

这个程序可以使用命令行参数传入两个数字，可以观察到使用时的差别，范例执行的结果如下。

```
>python decimal_demo.py 1.0 0.8
# 不使用 decimal
1.0 + 0.8 = 1.8
1.0 - 0.8 = 0.19999999999999996
1.0 * 0.8 = 0.8
1.0 / 0.8 = 1.25

# 使用 decimal
1.0 + 0.8 = 1.8
1.0 - 0.8 = 0.2
1.0 * 0.8 = 0.80
1.0 / 0.8 = 1.25
```

要注意的是，传入数字时必须使用字符串，而你也没有看错，`decimal.Decimal` 可以直接使用`+`、`-`、`*`、`/`等运算符，这样的方便性，就是 Python 在数值运算上大受欢迎的原因之一，运算的结果也是以 `decimal.Decimal` 类型返回。

提示>>>

在 Python 2 中，`1.0 - 0.8` 当然也存在误差，不过，`print(1.0 - 0.8)` 却会看到 0.2 的结果，这是因为 REPL 使用 `repr()` 函数来获取字符串描述并显示，而 `print()` 会使用 `str()` 来获取字符串描述进行显示，第 5 章会谈到 `repr()` 与 `str()`。

在乘法运算上，除了可以使用`*`进行两个数字的相乘之外，还可以使用`**`进行指数运算。例如：

```
>>> 2 ** 3
8
>>> 2 ** 5
32
>>> 2 ** 10
1024
>>> 9 ** 0.5
3.0
>>>
```

在除法运算上，有`/`与`//`两个运算符，前者若有小数部分会加以保留，后者的运算结果只会留下整数部分，在 Python 3 中，`/`一定产生浮点数，而`//`整数与整数运算会产生整数，如果是整数与浮点数进行`//`会产生浮点数。

```
>>> 10 / 3
3.3333333333333335
>>> 10 // 3
3
>>> 10 / 3.0
3.3333333333333335
>>> 10 // 3.0
3.0
>>>
```

提示>>>

Python 2 的“`/`”行为与 Python 3 不同，详情可以参考“Python 3 Tutorial 第二堂（2）数值与字符串类型”。

openhome.cc/Gossip/CodeData/PythonTutorial/NumericStringPy3.html

还有个%没谈到, $a \% b$ 时会进行除法运算并取余数作为结果。至于布尔值需要进行+、-、*、/等运算时, True 会被当成 1, False 会被当成 0, 接着再进行运算。

应用于字符串类型

使用+运算符可以串接字符串, 使用*可以重复字符串。

```
>>> text1 = 'Just'
>>> text2 = 'in'
>>> text1 + text2
'Justin'
>>> text1 * 10
'JustJustJustJustJustJustJustJustJust'
>>>
```

字符串不可变动, 因此+串接字符串时产生新字符串, 在程序设计语言强类型 (Strong type) 与弱类型 (Weak type) 的分类中, Python 偏向强类型, 也就是类型间在运算时一般不会自行发生类型转换。在 Python 中, 字符串与数字不能进行+运算, 若要进行数字和字符串的串接, 则要先将数字转换为字符串; 若要进行数字运算, 则要先将字符串解析为数字。例如:

```
>>> '10' + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> '10' + str(1)
'101'
>>> int('10') + 1
11
>>>
```

应用于列表与元组

列表有许多方面与字符串类似, 使用+运算符可以串接列表, 使用*可以重复列表。

```
>>> nums1 = ['one', 'two']
>>> nums2 = ['three', 'four']
>>> nums1 + nums2
['one', 'two', 'three', 'four']
>>> nums1 * 2
['one', 'two', 'one', 'two']
>>>
```

虽然列表本身的长度可变动, 不过用+串接两列表时, 实际上会产生新的列表, 然后将原有的两个列表中元素的引用复制到新产生的列表上, 同样的道理也适用于*重复列表时。请注意, 这里说的是复制引用, 而不是复制元素本身。可以用以下的实验来印证。

```
>>> nums1 = ['one', 'two']
>>> nums2 = ['three', 'four']
>>> nums_lt = [nums1, nums2]
>>> nums_lt
[['one', 'two'], ['three', 'four']]
>>> nums1[0], nums1[1] = '1', '2'
>>> nums_lt
[['1', '2'], ['three', 'four']]
>>>
```

在上例中, nums_lt[0]只是引用到 nums1 引用的列表, 因此在通过 nums1 来修改索引位置的元素之后, nums_lt 获取的元素也会是修改过的结果。

元组与列表有许多类似之处，+与*的操作在元组也有同样的效果。虽然说，元组本身的结构不可变动，不过这并不是指当中的元素也是不可变动的。例如：

```
>>> nums1 = ['one', 'two']
>>> nums2 = ['three', 'four']
>>> nums_tp = (nums1, nums2)
>>> nums_tp
(['one', 'two'], ['three', 'four'])
>>> nums1[0], nums1[1] = '1', '2'
>>> nums_tp
(['1', '2'], ['three', 'four'])
>>>
```

看到了吗？nums_tp 本身是个元组，然而放了两个列表作为元素，列表是可变动的，因此才会有这样的结果，所谓元组本身不能变动，是指不能执行 nums_tp[0] = ['five', 'six'] 这类的操作。

3.2.3 比较与赋值运算

对于大于、小于、等于这类的比较，Python 提供了 >、≥、<、≤、==、!=、<> 等运算符，其中 <> 效果与 != 相同，不过建议不要再用 <>，使用 != 比较清楚，<> 只是为了保持向后的兼容性 (Backwards compatibility) 而存在的。

这些比较运算有个很 Python 的特色，就是它们可以串接在一起，例如 $x < y \leq z$ ，其实相当于 $x < y$ and $y \leq z$ ，and 是布尔运算符，表示“而且”（或“与”）。如果愿意，也可以像 $w == x == y == z$ 这样一直串接下去。

请注意不要将 ==、!= 与 is 及 is not 搞混，==、!= 用来比较对象实际的值、状态等的相等性，而 is 及 is not 用来比较两个对象的引用是否相等。

想要知道对象是否能进行 >、≥、<、≤、==、!= 等比较，以及它们比较之后的结果为何，应该看它们的 __gt__()、__ge__()、__lt__()、__le__()、__eq__() 或 __comp__() 等方法如何实现，在第 5 章谈到自定义类时，就知道如何实现这些方法。

对于数值类型，进行 >、≥、<、≤、==、!= 等比较没有问题，就是比较数字。至于其他类型，字符串与列表也可以进行 >、≥、<、≤、==、!= 比较。字符串会逐字符按字典顺序进行比较，因此 'AAC' < 'ABC' 会是 True，'ACC' < 'ABC' 会是 False。读者可以编写一个简单的程序，通过命令行参数指定两个字符串，看看比较的结果。



operator compare.py

```
import sys

str1 = sys.argv[1]
str2 = sys.argv[2]

print("{}{} > {}{}? {}".format(str1, str2, str1 > str2))
print("{}{} == {}{}? {}".format(str1, str2, str1 == str2))
print("{}{} < {}{}? {}".format(str1, str2, str1 < str2))
```

执行结果如下：

```
>python compare.py Justin Monica
"Justin" > "Monica"? False
```

```
"Justin" == "Monica"? False
"Justin" < "Monica"? True
```

至于列表，则是逐元素进行比较，因[1, 2, 3] == [1, 2, 3]结果是 True，然而[1, 2, 3] > [1, 3, 3]结果是 False。


到目前为止只看过一个赋值运算符，也就是=这个运算符，事实上还有几个赋值运算符，如表 3.3 所示。

表 3.3 赋值运算符

赋值运算符	范例	结果
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
&=	a &= b	a = a & b
=	a = b	a = a b
^=	a ^= b	a = a ^ b
<<=	a <<= b	a = a << b
>>=	a >>= b	a = a >> b

3.2.4 逻辑运算

在逻辑上有所谓的“与”“或”和“非”，在 Python 中也提供了对应的逻辑运算符（Logical operator），分别为 and、or 和 not。下面这个程序是个简单的示范，可以判断命令行参数指定的两个字符串大小写的关系。

ab

```
operator  uppers.py

import sys

str1 = sys.argv[1]
str2 = sys.argv[2]

print('两个都大写?', str1.isupper() and str2.isupper())
print('有一个大写?', str1.isupper() or str2.isupper())
print('都不是大写?', not (str1.isupper() or str2.isupper()))
```

执行结果如下：

```
>python uppers.py Justin MONICA
两个都大写? False
有一个大写? True
都不是大写? False
```

在第 3.1.1 节中谈到布尔类型时曾经说过，将 None、False、0、0.0、0j、“”、()、[]、{}等传给 bool()都会返回 False，这些类型的其他值传入 bool()，则都会返回 True。在 and、or、not 运算中，遇到非 bool 类型时，也是这么判断。例如，如果 value 为 None、False、0、0.0、0j、“”、()、[]、{}。

等其中的一个，那么 `not value` 结果都会是 `True`。

`and`、`or` 有快捷运算的特性，`and` 的左操作数如果判定为假，就可以确认逻辑不成立，因此不用再继续运算右操作数了，`or` 则是左操作数判断为真，就可以确认逻辑成立，不用再运算右操作数。当判断确认时停留在哪个操作数，就返回该操作数的结果，例如：

```
>> [] and 'Justin'
[]
>>> [1, 2] and 'Justin'
'Justin'
>>> [] or 'Justin'
'Justin'
>>> [1, 2] or 'Justin'
[1, 2]
>>>
```

3.2.5 位运算

在数字设计上有 `AND`、`OR`、`NOT`、`XOR` 与反码运算，在 `Python` 中提供了对应的位运算符（Bitwise Operator），分别是 `&`（`AND`）、`|`（`OR`）、`^`（`XOR`）与 `~`（反码）。如果不会基本的位运算，可以从以下范例了解各个位运算的结果。

operator bitwise_demo.py

```
print('AND 运算: ')
print('0 AND 0 {:5d}'.format(0 & 0))
print('0 AND 1 {:5d}'.format(0 & 1))
print('1 AND 0 {:5d}'.format(1 & 0))
print('1 AND 1 {:5d}'.format(1 & 1))

print('\nOR 运算: ')
print('0 OR 0 {:6d}'.format(0 | 0))
print('0 OR 1 {:6d}'.format(0 | 1))
print('1 OR 0 {:6d}'.format(1 | 0))
print('1 OR 1 {:6d}'.format(1 | 1))

print('\nXOR 运算: ')
print('0 XOR 0 {:5d}'.format(0 ^ 0))
print('0 XOR 1 {:5d}'.format(0 ^ 1))
print('1 XOR 0 {:5d}'.format(1 ^ 0))
print('1 XOR 1 {:5d}'.format(1 ^ 1))
```

执行结果就是各个位运算的结果。

```
AND 运算:
0 AND 0    0
0 AND 1    0
1 AND 0    0
1 AND 1    1

OR 运算:
0 OR 0     0
0 OR 1     1
1 OR 0     1
1 OR 1     1

XOR 运算:
```



```

0 XOR 0 0
0 XOR 1 1
1 XOR 0 1
1 XOR 1 0

```

位运算是逐位运算，例如 10010001 与 01000001 进行 AND 运算，是一个位对应一个位的运算，答案就是 00000001。反码运算是将所有位 0 变 1，1 变 0。例如 00000001 经反码运算就会变为 11111110。例如：

```

>>> 0b10010001 & 0b01000001
1
>>> number1 = 0b0011
>>> number1
3
>>> ~number1
-4
>>> number2 = 0b1111
>>> number2
15
>>> ~number2
-16
>>>

```

number1 的 0011 经反码运算就变成 1100，这个数在计算机中用二补码¹来表示就是-4。要注意的是，Python 中的整数是有符号整数，当使用二进制写法表示一个整数时，例如用 0b1111 表示 15，实际上 1111 最左边的位是 0，经过反码运算后，1111 的部分会变成 0000，而最左边的位变成 1，整个值用二补码来表示就是-16。

在位运算上，Python 还有左移 (<<) 与右移 (>>) 两个运算符，左移运算符会将所有位往左移指定的位数，左边被挤出去的位会被丢弃，而右边补上 0；右移运算则是相反，会将所有位往右移指定的位数，右边被挤出去的位会被丢弃，而最左边补上原来的位，如果左边原来是 0 就补 0，1 就补 1。

下面使用左移运算来进行简单的 2 次方运算示范。

```
operator shift_demo.py
```

```

number = 1
print('2 的 0 次方: ', number);
print('2 的 1 次方: ', number << 1)
print('2 的 2 次方: ', number << 2)
print('2 的 3 次方: ', number << 3)

```

执行结果如下：

```

2 的 0 次方: 1
2 的 1 次方: 2
2 的 2 次方: 4
2 的 3 次方: 8

```

实际左移看看就知道为何可以如此进行次方的运算了。

```
00000001 → 1
```

¹ Two's complement: en.wikipedia.org/wiki/Two%27s_complement

```
00000010 → 2
00000100 → 4
00001000 → 8
```

实际上，&、|、^不仅能用在数值类型上，还可以应用在集合类型上，这是笔者觉得 Python 中最有趣也最实用的特性之一。例如，若想比较两个用户群组的状态，可以编写如下程序。



operator groups.py

```
import sys

admins = {'Justin', 'caterpillar'}
users = set(sys.argv[1:])
print('站长: {}'.format(admins & users))
print('非站长: {}'.format(users - admins))
print('全部用户: {}'.format(admins | users))
print('身份不重复的用户: {}'.format(admins ^ users))
print('站长群包括用户群? {}'.format(admins > users))
print('用户群包括站长群? {}'.format(admins < users))
```

在集合类型中，&可用于交集运算，|可用于并集运算，^可用于互斥运算。除此之外，上面的程序中也看到了，-可用于差集运算，>、<（以及≥、≤、==）可用于测试两个集合的包括关系。测试范例如下：

```
>python groups.py Justin Monica momor Irene hamimi
站长: {'Justin'}
非站长: {'Monica', 'momor', 'Irene', 'hamimi'}
全部用户: {'caterpillar', 'Monica', 'momor', 'Irene', 'Justin', 'hamimi'}
身份不重复的用户: {'caterpillar', 'momor', 'Irene', 'hamimi', 'Monica'}
站长群包括用户群? False
用户群包括站长群? False
```

在上面的范例中，可以看到 sys.argv[1:]这个程序代码，这是 Python 的切片（Slicing）运算，意思是将 sys.argv 索引 1 开始到列表尾端的全部元素切出成为一个新的列表，这是为了获取用户输入的全部命令行参数（因为 sys.argv[0]是.py 文件名），之后再交给 set()转换为集合类型。接下来我们将认识更多切片运算的方式。

3.2.6 索引切片运算

在 Python 的内建类型中，只要具有索引特性，基本上都能进行切片运算，像字符串、列表、元组等，下面以字符串为例来示范几个切片运算。

```
>>> name = 'Justin'
>>> name[0:3]
'Jus'
>>> name[3:]
'tin'
>>> name[:4]
'Just'
>>> name[:]
'Justin'
>>> name[:-1]
'Justi'
>>> name[-5:-1]
'usti'
```

```
>>>
```

上面示范的切片运算，可以是[start:end]形式，也就是指定起始索引（包括）与结尾索引（不包括）来切出子字符串。如果是[start:]形式，只指定起始索引，不指定结尾索引，表示切出从起始索引至字符串结束间的子字符串。若是[:end]形式，只指定结尾索引，不指定起始索引，表示切出从0索引至（不包括）结尾索引间的子字符串。如果两个都不指定，也就是[:]，就相当于复制字符串了。

Python 中的索引不仅可指定正值，还可以指定负值。实际上了解索引意义的开发人员都知道索引其实就是相对第一个元素的偏移值。在 Python 中，正值索引就是指正偏移值，负值索引就是负偏移值，-1 索引就是倒数第一个元素，-2 索引就是倒数第二个元素。

在切片运算时，起始索引与结尾索引也都可以指定负值。实际上，省略结尾索引时，就相当于结尾索引使用-1（省略起始索引时，就相当于使用0）。

切片运算的另一个形式是[start:end:step]，意思是切出起始索引与结尾索引（不包括）之间每次间隔 step 元素的内容（也就是省略 step 时，就相当于使用1）。例如：

```
>>> name = 'Justin'
>>> name[0:4:2]
'Js'
>>> name[2::2]
'si'
>>> name[:5:2]
'Jsi'
>>> name[::2]
'Jsi'
>>> name[::-1]
'nitsuJ'
>>>
```

注意最后一个范例，当 step 指定为正时，表示正偏移每 step 个取出元素，间隔指定为负时，表示负偏移每 step 个取出元素。[::-1]表示从索引0至结尾，以负偏移1方式获取字符串，结果就是反转字符串了。

以上的操作对于元组也是适用的，来看几个简单的例子。

```
>>> nums[0:3]
(10, 20, 30)
>>> nums[1:]
(20, 30, 40, 50)
>>> nums[:4]
(10, 20, 30, 40)
>>> nums[-5:-1]
(10, 20, 30, 40)
>>> nums[::-1]
(50, 40, 30, 20, 10)
>>>
```

在使用[:]时要注意，[:]只是作浅层复制（Shallow copy），也就是复制元素时只复制元素的引用，而不是复制整个元素。对于字符串来说，这不会造成什么困扰，不过若是元组或列表中包含可变动元素，就要注意了。

```
>>> nums1 = [10, 20, 30, 40, 50]
>>> nums2 = [60, 70, 80, 90, 100]
>>> t1p1 = (nums1, nums2)
>>> t1p2 = t1p1[:]
```



```
>>> tlp2[0][0] = 1
>>> tlp1
([1, 20, 30, 40, 50], [60, 70, 80, 90, 100])
>>>
```

由于只复制元素的引用，因此在上面的范例中，修改了列表 `tlp2` 索引 0 的内容，通过 `tlp1` 也就看到了修改的结果。

若是列表这类元素可变动的结构，在进行切片运算时，则可以取代元素，例如：

```
>>> lt = ['one', 'two', 'three', 'four']
>>> lt[1:3] = [2, 3]
>>> lt
['one', 2, 3, 'four']
>>> lt[1:3] = ['ohoh']
>>> lt
['one', 'ohoh', 'four']
>>> lt[:] = []
>>> lt
[]
```

事实上，与其说是取代元素，不如说是将指定索引范围的元素清除掉，再将指定列表的元素放入。因此 `lt[1:3] = ['ohoh']` 运算之后，有两个元素消失了，只置入了一个 'ohoh'，而对于 `lt[:] = []` 这样的赋值，就相当于清空元素了（实际上是让 `lt` 引用至 `[]`）。

对于可变动的列表，若要直接删除某一段元素，也可以使用 `del` 结合切片运算。例如：

```
>>> lt = ['one', 'two', 'three', 'four']
>>> del lt[1:3]
>>> lt
['one', 'four']
>>>
```

3.3 重点复习

在 Python 中，所有的数据都是对象，可以使用直接表示法来编写一些内建类型。想知道某个数据的类型，可以使用 `type()` 函数。

从 Python 3 之后，整数类型为 `int`，不再区分整数与长整数（在 Python 2.x 中分别是 `int` 与 `long` 两种类型），整数的长度不受限制（除了硬件物理上的限制之外）。

如果有个字符串是 '3.14'，想要获取整数部分并解析为 `int`，不可以直接使用 `int()`，这样会出现 `ValueError` 错误，而要先使用 `float()` 解析为 `float`，接着再用 `int()` 获取 `int` 整数。

将 `None`、`False`、`0`、`0.0`、`0j`（复数）、`"`（空字符串）、`()`（空元组）、`[]`（空列表）、`{}`（空字典）等传给 `bool()`，都会返回 `False`，若这些类型的其他值传入 `bool()`，则都会返回 `True`。

Python 支持复数的直接表示法，编写时使用 `a + bj` 的形式，复数是 `complex` 类的实例，可以直接对复数进行数值运算。

可以使用 `"或"` 括住文字，两者在 Python 中具有相同的作用，在 Python 3 之后的版本都是产生 `str` 实例，可视情况互换。想使用原始字符串（Raw String）来表示，只要在字符串前加上 `r` 即可。

如果字符串内容必须跨越数行，可以使用三重引号，在三重引号之间输入的任何内容，最后的字符串会照单全收，像换行、缩排等。

`print()`函数的显示默认会换行，`print()`有个 `end` 参数，在指定的字符串显示之后，`end` 参数指定的字符串才会输出。

目前的 Python 3 支持两种格式化方式，一种旧式（从 Python 2 就存在），一种新式（从 Python 2.6、2.7 开始支持）。如果使用 Python 3 之后的版本（或者 Python 2.6、2.7），建议使用新的格式化方式。

从 Python 3 之后，每个字符串都包含了 Unicode 字符，每个字符串都是 `str` 类型。如果想将字符串转为指定的编码，可以使用 `encode()` 方法指定编码，得到的结果是一个 `bytes` 实例，当中包含了字节数据。若有一个 `bytes` 实例，也可以使用 `decode()` 方法，指定该字节代表的编码，将 `bytes` 编码为 `str` 实例。

可以在字符串前加上个 `b` 来创建一个 `bytes`，这是从 Python 3.3 之后开始支持的语法。类似的，也可以在字符串前加上一个 `u`，结果就是一个 `str`。在 Python 3 之后，字符串默认就是 `str`，因此不用特别加上 `u`，这个语法是为了增加与 Python 2 的兼容性而增加的。

集合中的元素必须都是 `hashable`（可哈希运算的）。

创建字典时，每个键用来获取对应的值，字典中的键不重复，必须是 `hashable`（可哈希运算的）。

如果要获取字典中的每一对键与值，可以使用 `items()` 方法，以 `dict_items` 对象通过元组获取每一对键与值。只要获取键，就可以使用 `keys()` 方法返回 `dict_keys` 对象，然后可以逐一获取每个键。如果要获取值，则可以使用 `values()` 方法来返回 `dict_values` 对象。

元组在许多地方都很像列表，是有序的结构，可以使用 `[]` 指定索引获取元素，不过元组创建之后，就不能变动了。

Python 3 之后，还进一步扩展了拆解元素的功能，称为 `Extended Iterable Unpacking`（扩展可迭代拆解）。

Python 属于动态类型语言，变量本身并没有类型信息，变量始终是个引用至实际对象的名称，赋值运算只是改变了变量的引用对象，同一个变量可以前后指定不同的数据类型，若想通过变量操作对象的某个方法，只要确认该对象上确实有该方法即可。

动态类型语言界流行的鸭子类型（`Duck typing`）：“如果它走路像个鸭子，游泳像个鸭子，叫声像个鸭子，那它就是鸭子。”

开发人员基本上都要知道 IEEE 754 浮点数算术标准¹，Python 也遵守此标准，这个算术标准不使用小数点，而是使用分数及指数来表示小数。

如果需要处理小数，而且需要精确的结果，那么可以使用 `decimal.Decimal` 类。`decimal.Decimal` 可以直接使用 `+`、`-`、`*`、`/` 等运算符。

在强数据类型与弱数据类型的分类中，Python 偏向强类型，也就是类型间在运算时，一般不会自行发生类型转换。在 Python 中，字符串与数字不能进行 `+` 运算，若要进行数字和字符串的串接，则要先将数字转换为字符串；若要进行数字运算，则要先将字符串解析为数字。

¹ IEEE Standard for Floating-Point Arithmetic: en.wikipedia.org/wiki/IEEE_floating_point

+串接两个列表，实际上会产生新的列表，然后将原有的两个列表中的元素引用，复制至新产生的列表上，同样的道理也适用于*重复列表时的复制引用，而不是复制元素本身。

◇效果与!=相同，不过建议不要再用◇，使用!=比较清楚，◇只是为了保持向后的兼容性（Backwards compatibility）而存在的。

比较运算有个很 Python 的特色，就是它们可以串接在一起，例如 $x < y \leq z$ ，其实相当于 $x < y$ and $y \leq z$ 。

and、or 有快捷运算的特性，and 的左操作数若判定为假，就可以确认逻辑不成立，因此不用再继续运算右操作数，or 则是左操作数判断为真，就可以确认逻辑成立，不用再运算右操作数。当判断确认时停留在哪个操作数，就会返回该操作数的值。

&、|、^不仅仅能用在数值类型上，还可以应用在集合类型上。

在 Python 的内建类型中，只要具有索引特性，基本上都能进行切片运算。Python 中的索引不仅可指定正值，还可以指定负值。

在使用[:]时要注意，[:]只是作浅层复制（Shallow copy），也就是复制元素时只复制元素的引用，而不是复制整个元素。

课后练习

实践题

1. 编写一个程序，可用命令行参数接受用户输入的字符串列表，列出列表中不重复的字符串与数量。例如有以下的执行结果：

```
>python exercise1.py your right brain has nothing left and your left brain has nothing right
有 7 个不重复的字符串：{'left', 'has', 'your', 'right', 'nothing', 'brain', 'and'}
```

2. 编写一个程序，可用命令行参数接受用户输入的字符串列表，第一个参数可用来指定查询后续参数中某字符串出现的次数。例如有以下的执行结果：

```
>python exercise2.py brain your right brain has nothing left and your left brain has nothing right
brain 出现了 2 次。
```

提示 >>>

list、str、tuple（元组）等类型，都有个 count()方法，详情可参阅：
docs.python.org/3.5/library/stdtypes.html#common-sequence-operations。

流程语句与函数

学习目标

- 认识基本流程语句
- 使用 for Comprehension
- 认识函数与变量作用域
- 运用一级函数特性
- 使用 yield 创建生成器



4.1 流程语句

现实生活中待解决的事务千奇百怪，在计算机发明之后，想要使用计算机解决的需求也是各种各样：“如果”发生了……就要……；“对于”……就一直执行……；“如果”……就“中断”……。为了告诉计算机在特定条件下该执行的操作，要使用各种条件判断语句来定义程序执行的流程。

4.1.1 if 分支判断

流程语句中最简单也最常见的就是 if 分支判断语句，在 Python 中是这样写的：

basic hello.py

```
import sys

name = 'Guest'
if len(sys.argv) > 1:
    name = sys.argv[1]
print('Hello, {}'.format(name))
```

在 Python 中，一个程序区块是使用冒号 (:) 开头，之后同一区块范围要有相同的缩排，不可混用不同空格数量，不可混用空格与 Tab，Python 的建议是使用 4 个空格作为缩排。

这个范例中，默认的名称是 'Guest'，如果执行时提供了命令行参数，则 sys.argv 的长度就会大于 1（记得索引 0 是 .py 文件名），len(sys.argv) > 1 的结果是 True，if 条件成立，因而将 name 设置为用户提供的命令行参数。范例的执行如下：

```
>python hello.py
Hello, Guest

>python hello.py Justin
Hello, Justin
```

if 可以搭配 else，在 if 条件不成立时，执行 else 中定义的程序代码，例如编写一个判断数字为奇数或偶数的范例：



basic is_odd.py

```
import sys

number = int(sys.argv[1])
if number % 2:
    print('{} 为奇数'.format(number))
else:
    print('{} 为偶数'.format(number))
```

如果是偶数，那么 number % 2 就会是 0，在 if 判断式中就会被认定为 False，因此会执行 else 区块的程序语句。范例执行结果如下：

```
>python is_odd.py 10
```


10 为偶数

```
>python is_odd.py 9
9 为奇数
```

Python 的区块定义方式可以避免 C/C++、Java 这类 C-like 语言某些不明确的情况。例如在 C-like 语言中，可能会出现这样的程序代码：

```
if(condition1)
    if(condition2)
        doSomething();
else
    doOther();
```

乍看之下，else 似乎与第一个 if 配对，但实际上，else 是与最近的 if 配对，也就是第二个 if。在 Python 中的区块定义就没有这个问题。

```
if condition1:
    if condition2:
        do_something()
else:
    do_other()
```

以上例而言，else 必定与第一个 if 配对，如果是下例，那么 else 必定是与第二个 if 配对。

```
if condition1:
    if condition2:
        do_something()
    else:
        do_other()
```

如果有多重判断，那么可以使用 if..elif..else 结构。例如：

basic grade.py

```
score = int(input('输入分数: '))
if score >= 90:
    print('得 A')
elif 90 > score >= 80:
    print('得 B')
elif 80 > score >= 70:
    print('得 C')
elif 70 > score >= 60:
    print('得 D')
else:
    print('不及格')
```

范例执行结果如下：

```
>python grade.py
输入分数: 88
得 B
```

在 Python 中有个 if..else 表达式语句。直接来看怎么用来改写之前 is_odd.py 的程序代码。

basic is_odd2.py

```
import sys
```

```
number = int(sys.argv[1])
print('{} 为 {}'.format(number, '奇数' if number % 2 else '偶数'))
```

当 if 的条件式成立时，会返回 if 前的数值，若不成立则返回 else 后的数值，这个程序的执行结果与 is_odd.py 是相同的。

4.1.2 while 循环

Python 提供了 while 循环，可根据指定条件判断式来判断是否执行循环体，语句格式如下所示。

```
while 条件式:
    语句
else:
    语句
```

在条件判断式成立时，会执行 while 区块，至于可与 while 搭配的 else，是其他程序设计语言几乎没有的特色，不过基本上不建议使用，原因稍后再来说明。先来看个很无聊的游戏，看谁可以最久不碰到 5 这个数字。

basic lucky5.py

```
import random

number = 0
while number != 5:  ← ①如果不是 5 就执行循环
    number = random.randint(0, 9)  ← ②随机产生 0 到 9 的数
    print(number)
    if number == 5:
        print('我碰到 5 了....Orz')
```

这个范例的 while 判断式会判断 number 是否为 5①，如果判断为 True 就执行循环，random.randint()指定了随机产生 0 到 9 的整数②。范例的执行结果可能如下（因为有随机数的产生，所以每次结果不一样）。

```
4
5
我碰到 5 了....Orz
```

至于可以和 while 搭配的 else，乍看会以为类似 if...else，误认为如果没有执行 while 循环，就执行 else 的部分，然而实际上是 while 循环正常执行结束，也会执行 else 的部分。

```
>>> while False:
...     print('while')
... else:
...     print('else')
...
else
>>> while num == 0:
...     print('while')
...     num = 1
... else:
...     print('else')
...
while
else
```

>>>

若不想让 `else` 执行，必须是 `while` 中因为 `break` 而中断循环，下面是求最大公因数的程序，程序代码经过特别安排，或许比较好懂这个程序的逻辑。

```
basic gcd.py
```

```
print('输入两个数字...')

m = int(input('数字 1: '))
n = int(input('数字 2: '))

while n != 0:
    r = m % n
    m = n
    n = r
    if m == 1:
        print('互质')
        break      ← break 可用来中断循环
else:
    print("最大公因数: ", m)
```

在上面的范例中，如果求出的最大公因数是 1，显示两数互质并使用 `break`，在循环中若遇到了 `break`，循环就会中断，此时就不会执行 `else`。范例的执行结果如下。

```
>python gcd.py
输入两个数字...
数字 1: 20
数字 2: 16
最大公因数: 4

>python gcd.py
输入两个数字...
数字 1: 10
数字 2: 3
互质
```

在范例程序代码中，特别使用粗体标示的部分，活像组成了一对 `if...else`，“if 某条件而执行 `break` 了，就不会执行 `else`”，或者反过来想“if 没有执行 `break`，就执行 `else`”，这样或许会比较能理解 `while` 与 `else` 的关系吧！

无论如何，这实在太难懂了，建议别使用 `while` 与 `else` 的形式，上面的范例，改成以下写法才容易理解。

```
basic gcd2.py
```

```
print('输入两个数字...')

m = int(input('数字 1: '))
n = int(input('数字 2: '))

while n != 0:
    r = m % n
    m = n
    n = r

if m == 1:
```



```

    print('互质')
else:
    print("最大公因数: ", m)

```

4.1.3 for in 迭代

若要按序迭代某个序列，例如字符串、列表、元组（tuple），则可以使用 for in 语句。例如，迭代用户提供的命令行参数，再转为大写后输出。

basic uppers.py

```

import sys
for arg in sys.argv:
    print(arg.upper())

```

要被迭代的序列放在 in 之后，对于字符串、列表、元组等具有索引特性的序列，for in 会按照索引顺序逐一取出元素，并赋值给 in 之前的变量。范例的执行结果如下：

```

>python uppers.py justin monica irene
UPPERS.PY
JUSTIN
MONICA
IRENE

```

如果在迭代的同时需要同时提供索引信息，那么有几个方式，例如使用 range() 函数产生一个指定的数字范围，使用 for in 进行迭代，再利用迭代的数字作为索引。例如：

```

>>> name = 'Justin'
>>> for i in range(len(name)):
...     print(i, name[i])
...
0 J
1 u
2 s
3 t
4 i
5 n
>>>

```

range() 函数的形式是 range(start, stop[, step])，start 省略时默认是 0，step 是步进值，省略时默认是 1，因此在上例中，range(len(name)) 用于产生 0 到 5 的数字。

也可以使用 zip() 函数，将两个序列的各个元素像拉链般一对一配对（这就是为什么它叫 zip 的原因，实际上 zip() 可以用于接受多个序列），产生一个新的列表，当中每个元素都是元组，包括了配对后的元素。

```

>>> list(zip([1, 2, 3], ['one', 'two', 'three']))
[(1, 'one'), (2, 'two'), (3, 'three')]
>>>

```

zip() 函数会返回一个 zip 对象，这个对象实际上还不包括真正配对后的元素，也就是具有惰性求值的特性（range() 产生的 range 对象也是）。zip 对象可以使用 for in 进行迭代，因此若迭代时需要索引信息，可以如下编写：

```

name = 'Justin'

```

```
for i, c in zip(range(len(name)), name):
    print(i, c)
```

在这里还使用了元组拆解的特性，将每一对元组中的元素拆解再赋值给 `i` 与 `c` 变量。

实际上，若真的要迭代时具有索引信息，建议使用 `enumerate()` 函数而不是 `range()` 函数，`enumerate()` 会返回 `enumerate` 对象，一样具有惰性求值特性，且可使用 `for in` 进行迭代，`enumerate` 可用于获取元组元素，例如：

```
>>> name = 'Justin'
>>> list(enumerate(name))
[(0, 'J'), (1, 'u'), (2, 's'), (3, 't'), (4, 'i'), (5, 'n')]
>>>
```

因此，迭代时具有索引信息，也可以使用以下方式：

```
name = 'Justin'
for i, c in enumerate(name):
    print(i, c)
```

在默认的情况下，`enumerate()` 会从 0 开始计数，如果想要从其他数字开始，可由 `enumerate()` 的第二个参数来指定。例如从 1 开始：

```
name = 'Justin'
for i, c in enumerate(name, 1):
    print(i, c)
```

实际上，在之后章节中读者会看到，只要是实现了 `__iter__()` 方法的对象，都可以通过 `__iter__()` 方法返回一个迭代器 (Iterator)，这个迭代器可以使用 `for in` 来迭代，像之前的 `range`、`zip`、`enumerate` 对象就是如此。

集合也实现了 `__iter__()` 方法，因此可以进行迭代，不过因为集合是无序的，所以只能迭代出元素，但不一定是我们想要的顺序；至于想要迭代字典键与值，可以使用它的 `keys()`、`values()` 或 `items()` 方法，它们分别会返回 `dict_keys`、`dict_values`、`dict_items` 对象，而且都实现了 `__iter__()` 方法，因此也可以使用 `for in` 来迭代。举例来说，同时迭代字典的键与值：

```
>>> passwds = {'Justin' : 123456, 'Monica' : 54321}
>>> for name, passwd in passwds.items():
...     print(name, passwd)
...
Justin 123456
Monica 54321
>>>
```

因为 `dict_items` 的元素是元组，包括了一对键和值，同样地，这里使用了元组拆解的特性，将元组的键、值拆解给 `name` 与 `passwd` 变量。如果直接对字典进行 `for in` 迭代，默认会进行键的迭代。

类似 `while` 可与 `else` 配对，`for in` 也有个与 `else` 配对的形式，若不想让 `else` 执行，必须是 `for in` 中因为 `break` 而中断迭代，不过建议不要使用 `for in...else` 的形式，如果真的想看个应用，下面有一个例子，可用来判断指定的数字是否为质数。

```
basic is_prime.py
```

```
number = int(input('输入数字: '))
half = number // 2
for num in range(2, half + 1):
    if number % num == 0:
```

```

    print(number, '不是质数')
    break      ← break 可用来中断遍历
else:
    print(number, '是质数')

```

4.1.4 pass、break、continue

有时在某个区块中，并不想执行任何程序语句，或者稍后会编写些什么程序语句，对于还没打算编写任何语句的区块，可以先放个 `pass`。例如：

```

if is_prime:
    print('找到质数')
else:
    pass

```

`pass` 就真的是 `pass`（即通过），什么都不做，只是用来维持程序代码结构的完整性。虽然如此，未来也许会常常用到它，因为我们可能经常要做一些小测试，或者是先执行一下程序，看看其他已编写好的程序代码是否如期运行，这时 `pass` 就会派上用场。

至于 `break`，在之前介绍 `while` 与 `for in` 时已经知道了它的功能，分别可用来中断 `while` 循环、`for in` 的迭代。在这里再提一次，是为了与 `continue` 对照。在 `while` 循环中遇到 `continue`，不执行此次循环后续的程序代码，直接进行下一次的循环，在 `for in` 迭代遇到 `continue`，不执行此次迭代后续的程序代码，直接进行下一次迭代。

以下是利用 `continue` 的特性实现一个只显示小写字母的程序。

basic show_uppers.py

```

text = input('输入一个字符串: ')
for letter in text:
    if letter.isupper():
        continue
    print(letter, end='')

```

这个范例在遇到大写字母时就会执行 `continue`，因此该次不会执行 `print()`。范例执行如下：

```

>python show_uppers.py
输入一个字符串: This is a Question!
his is a uestion!

```

4.1.5 for Comprehension

如果用户输入的命令行参数是数字，若想要将这些数字全部进行平方运算，该怎么做呢？也许会想出这样的写法：

```

import sys

squares = []
for arg in sys.argv[1:]:
    squares.append(int(arg) ** 2)
print(squares)

```

将一个列表转为另一个列表，是很常见的操作，Python 对这类的需求提供了 `for Comprehension`

语句，可以如下实现需求：

basic square.py

```
import sys

squares = [int(arg) ** 2 for arg in sys.argv[1:]]
print(squares)
```

对于 `for arg in sys.argv[1:]` 这部分，其作用是逐一迭代出命令行参数并赋值给 `arg` 变量，之后执行 `for` 左方的 `int(arg) ** 2` 运算，使用 `[]` 括起来，表示每次迭代的运算结果，会被收集为一个列表。执行结果如下：

```
>python square.py 10 20 30
[100, 400, 900]
```

`for Comprehension` 也可以与条件判断式结合，这可以构成一个过滤的功能。例如想收集某个列表中的奇数元素至另一个列表，在不使用 `for Comprehension` 时，可以如下编写。

```
import sys

odds = []
for arg in sys.argv[1:]:
    if int(arg) % 2:
        odds.append(arg)
print(odds)
```

若使用 `for Comprehension`，则可以改写为以下的程序代码。

basic odds.py

```
import sys

odds = [arg for arg in sys.argv[1:] if int(arg) % 2]
print(odds)
```

在这个例子中，只有在 `if` 条件式成立时，`for` 左边的表达式才会被执行，并收集为最后结果列表中的元素。执行结果如下：

```
>python odds.py 11 8 9 5 4 6 3 2
['11', '9', '5', '3']
```

如果要形成嵌套结构也是可行的，不过建议别太过火，不然可读性会迅速降低。简单地将矩阵转换为二维的列表倒还不错。

```
>>> matrix = [
...     [1, 2, 3],
...     [4, 5, 6],
...     [7, 8, 9]
... ]
>>> array = [element for row in matrix for element in row]
>>> array
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

另一个例子是使用 `for Comprehension` 来得到两个序列的排列组合。

```
>>> [letter1 + letter2 for letter1 in 'Justin' for letter2 in 'momor']
```

```
['Jm', 'Jo', 'Jm', 'Jo', 'Jr', 'um', 'uo', 'um', 'uo', 'ur', 'sm', 'so', 'sm', 'so', 'sr', 'tm',
'to', 'tm', 'to', 'tr', 'im', 'io', 'im', 'io', 'ir', 'nm', 'no', 'nm', 'no', 'nr']
>>>
```

在 for Comprehension 两边放上[], 表示会产生列表, 如果数据源很长, 或者数据源本身是个有惰性求值特性的生成器时, 直接产生列表显得没有效率, 这时可以在 for Comprehension 两旁放上(), 这样就会创建一个 generator (生成器) 对象, 具有惰性求值特性。

举个例子来说, Python 中有个 sum() 函数, 可以计算指定序列的数字加总值, 若调用 sum([1, 2, 3]), 则结果为 6。如果想计算 1 到 10000 的加总值呢? 使用 sum([n for n in range(1, 10001)]) 是可以达到目的, 不过, 这会先产生具有 10000 个元素的列表, 然后再交给 sum() 函数运算, 此时可以写成 sum(n for n in range(1, 10001)), 这样就不会有产生列表的负担。

这其实也在说明, 只写 n for n in range(1, 10001) 就是一个生成器表达式了, 因此在传给 sum() 函数时, 不必再写成 sum((n for n in range(1, 10001))), 再加上括号的情况。就是直接引用一个生成器, 例如 g = (n for n in range(1, 10001)) 的情况。

for Comprehension 也可以用来创建集合, 只要在 for Comprehension 两边放上 {}。例如创建一个集合, 其中包括了源字符串中不重复的大写字母。

```
>>> text = 'Your Right brain has nothing Left. Your Left brain has nothing Right'
>>> {c for c in text if c.isupper()}
{'Y', 'R', 'L'}
>>>
```

若想使用 for Comprehension 来创建字典实例, 也是可行的。例如:

```
>>> names = ['Justin', 'Monica', 'Irene']
>>> passwds = [123456, 654321, 13579]
>>> {name : passwd for name, passwd in zip(names, passwds)}
{'Justin': 123456, 'Irene': 13579, 'Monica': 654321}
>>>
```

上面的 zip 函数将 names 与 passwds 两两相扣结合在一起成为元组, 每个元组中的一对元素, 会在 for Comprehension 中拆解并赋值给 name 与 passwd, 最后 name 与 passwd 组成字典的每一对键和值。

那么可以使用 for Comprehension 创建元组吗? 可以的, 不过不是在 for Comprehension 两旁放上(), 这样就会创建一个 generator (生成器) 对象, 而不是元组, 想要用 for Comprehension 创建元组, 可以将 for Comprehension 生成器表达式传给 tuple()。例如:

```
>>> tuple(n for n in range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>>
```

4.2 定义函数

在学会了流程语句之后, 读者应该能编写一些小程序了, 根据不同的条件计算出不同的结果。然而, 可能会发现有些程序片段你一用再用, 老是复制、粘贴、修改变量名称, 让程序代码显得笨拙而且不易维护, 这时就可以将可复用 (即可重复使用) 的程序片段定义为函数, 之后直接调用函数来复用这些程序片段。

4.2.1 使用 def 定义函数

为了复用某个程序片段而开始复制、粘贴、修改变量名称时，或者发现两个或多个程序片段极为类似，只有当中几个计算用到的数值或变量不同时，才可以考虑将这些程序片段定义为函数。例如发现程序中：

```
# 其他程序片段...
max1 = a if a > b else b
# 其他程序片段...
max2 = x if x > y else y
# 其他程序片段...
```

这时可以定义函数来封装程序片段，将程序片段中引用不同数值或变量的部分设计为参数，例如：

```
def max(num1, num2):
    return num1 if num1 > num2 else num2
```

定义函数时要使用 `def` 关键词，`max` 是函数名称，`num1`、`num2` 是参数名称，若要返回值，则可以使用 `return`。如果函数执行完毕但没有使用 `return` 返回值，或者使用了 `return` 结束函数但没有指定返回值，默认就会返回 `None`。

这么一来，原先的程序片段就可以修改为：

```
max1 = max(a, b)
# 其他程序片段...
max2 = max(x, y)
# 其他程序片段...
```

函数是一种抽象，对程序片段的抽象，在定义了 `max` 函数之后，客户端对求最大值的程序片段被抽象为 `max(x, y)` 这样的函数调用，求值程序片段的实现被隐藏了起来。

函数也可以调用自身，这被称为递归（Recursion），举个例子来说，4.1.2 小节中的 `gcd2.py` 求最大公因数的程序片段，若定义为函数且用递归求解，可以写成：

```
func gcd.py

def gcd(m, n):
    if n == 0:
        return m
    else:
        return gcd(n, m % n)

print('输入两个数字...')

m = int(input('数字 1: '))
n = int(input('数字 2: '))

r = gcd(m, n)
if r == 1:
    print('互质')
else:
    print("最大公因数: ", r)
```


提示》》

有不少人觉得递归很复杂，其实只要一次只处理一个任务，而且每次递归只专注当前的子任务，递归其实反而清晰易懂，像这里的 `gcd()` 函数可清楚地看出辗转相除法的定义。如果有兴趣，那么也可以参考我在《递归的美丽与哀愁》中的一些想法：

openhome.cc/Gossip/Programmer/Recursive.html

在 Python 中，函数中还可以定义函数，称为局部函数（Local function），可以使用局部函数将某函数中的算法组织为更小的单元，例如在选择排序的实现时，每次会从未排序部分选择一个最小值放到已排序部分之后，在下面的范例中寻找最小值的索引时，就是以局部函数的方式实现的。



func sele_sort.py

```
import sys

def sele_sort(number):
    # 找出未排序中最小值
    def min_index(left, right):
        if right == len(number):
            return left
        elif number[right] < number[left]:
            return min(right, right + 1)
        else:
            return min(left, right + 1)

    for i in range(len(number)):
        selected = min_index(i, i + 1)
        if i != selected:
            number[i], number[selected] = number[selected], number[i]

number = [int(arg) for arg in sys.argv[1:]]
sele_sort(number)
print(number)
```

可以看到，局部函数的好处之一就是可以直接存取包含它的外部函数的参数或先前声明的局部变量，如此可减少调用函数时参数的传递。执行结果如下：

```
>python sele_sort.py 1 3 2 5 9 7 6 8
[1, 2, 3, 5, 7, 6, 8, 9]
```

4.2.2 参数与自变量

在 Python 中不支持函数重载（Overload）的实现，也就是在同一个命名空间中不能有相同的函数名称。如果定义了两个函数具有相同的名称，但拥有不同参数个数，那么之后定义的函数会覆盖先前定义的函数。例如：

```
>>> def sum(a, b):
...     return a + b
...
>>> def sum(a, b, c):
...     return a + b + c
...
>>> sum(1, 2)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum() missing 1 required positional argument: 'c'
>>>
```

在上面的例子中，因为后来定义的 `sum()` 有三个参数，它会覆盖先前定义的 `sum()`，如果只指定两个参数，就会引发 `TypeError` 错误。实际上，第一次自行定义 `sum()` 时，也覆盖了标准链接库内建的 `sum()` 函数。

参数默认值

虽然不支持函数重载的实现，不过在 Python 中可以使用默认自变量来有限度地模仿函数重载。例如：

```
def account(name, number, balance = 100):
    return {'name': name, 'number': number, 'balance': balance}

# 显示 {'name': 'Justin', 'balance': 100, 'number': '123-4567'}
print(account('Justin', '123-4567'))
# 显示 {'name': 'Monica', 'balance': 1000, 'number': '765-4321'}
print(account('Monica', '765-4321', 1000))
```

使用参数默认值时，必须小心指定了可变动对象时的一个陷阱，Python 在执行到 `def` 时，就会按照定义创建相关的资源。来看看下面会有什么问题。

```
>>> def prepend(elem, lt = []):
...     lt.insert(0, elem)
...     return lt
...
>>> prepend(10)
[10]
>>> prepend(10, [20, 30, 40])
[10, 20, 30, 40]
>>> prepend(20)
[20, 10]
>>>
```

在上例中，`lt` 默认值设置为 `[]`，由于 `def` 是条语句，执行到 `def` 的函数定义时就创建了 `[]`，而这个列表对象会一直存在，如果没有指定 `lt`，使用的就会一直是一开始指定的列表对象，因此随着每次调用都不指定 `lt` 的值，前置的目标列表都是同一个列表。

想要避免这样的问题，可以将 `prepend()` 的 `lt` 参数默认值设为 `None`，并在函数中指定真正的默认值。例如：

```
>>> def prepend(elem, lt = None):
...     lt = [] if lt is None else lt
...     lt.insert(0, elem)
...     return lt
...
>>> prepend(10)
[10]
>>> prepend(10, [20, 30, 40])
[10, 20, 30, 40]
>>> prepend(20)
[20]
>>>
```

在上面的 `prepend()` 函数中, `lt` 为 `None` 时使用 `[]` 创建新的列表实例, 这样就不会有之前的问题。

🔗 关键词自变量

事实上, 在调用函数时, 并不一定要按参数声明顺序来传入自变量, 可以指定参数名称来设置其自变量值, 它被称为关键词自变量。例如:

```
def account(name, number, balance):
    return {'name': name, 'number': number, 'balance': balance}

# 显示 {'name': 'Monica', 'balance': 1000, 'number': '765-4321'}
print(account(balance = 1000, name = 'Monica', number = '765-4321'))
```

🔗 *与**

如果有个函数拥有固定的参数, 而我们有一个序列, 像列表、元组, 只要在传入时加上 `*`, 列表或元组中各个元素就会自动拆解给各个参数。例如:

```
def account(name, number, balance):
    return {'name': name, 'number': number, 'balance': balance}

# 显示 {'name': 'Justin', 'balance': 1000, 'number': '123-4567'}
print(account(*('Justin', '123-4567', 1000)))
```

像 `sum()` 这种加总数字的函数, 事先无法预期要传入的自变量个数, 可以在定义函数的参数时使用 `*`, 表示该参数接受不定长度的自变量。例如:

```
def sum(*numbers):
    total = 0
    for number in numbers:
        total += number
    return total

print(sum(1, 2))      # 显示 3
print(sum(1, 2, 3))   # 显示 6
print(sum(1, 2, 3, 4)) # 显示 10
```

传入函数的自变量, 会被收集到一个元组中, 再设置给 `numbers` 参数, 这适用于参数个数不固定而且会按序迭代处理参数的场合。

如果有个字典, 打算按照键的名称赋值给对应的参数名称, 可以在字典前加上 `**`, 这样字典中各对键和值就会自动拆解给各个参数。例如:

```
def account(name, number, balance):
    return {'name': name, 'number': number, 'balance': balance}

params = {'name': 'Justin', 'number': '123-4567', 'balance': 1000}
# 显示 {'name': 'Justin', 'balance': 1000, 'number': '123-4567'}
print(account(**params))
```

如果参数个数越来越多, 而且每个参数名称都有其意义, 像 `def ajax(url, method, contents, datatype, accept, headers, username, password)`, 这样的函数定义不但“丑陋”, 调用时也很麻烦, 单纯只搭配关键词自变量或默认自变量, 也不见得能改善多少, 将来若因为需求而必须增减参数时, 也会影响函数的调用者, 势必要逐一修改影响到的程序, 造成未来程序扩充时的麻烦。

这个时候可以试着使用 `**` 来定义参数, 让指定的关键词自变量收集为一个字典。例如:


```
def ajax(url, **user_settings):
    settings = {
        'method' : user_settings.get('method', 'GET'),
        'contents' : user_settings.get('contents', ''),
        'datatype' : user_settings.get('datatype', 'text/plain'),
        # 其他设置 ...
    }
    print('请求 {}'.format(url))
    print('设置 {}'.format(settings))

ajax('http://openhome.cc', method = 'POST', contents = 'book=python')
my_settings = {'method' : 'POST', 'contents' : 'book=python'}
ajax('http://openhome.cc', **my_settings)
```

像这样定义函数就显得优雅许多，调用函数时可使用关键词自变量，在函数内部也可实现默认自变量的效果，这样的设计在未来程序扩充时比较有利，因为如果需要增减参数，那么只需修改函数的内部实现，所以函数的调用者不会受到影响。

在上面的函数定义中假设 `url` 为每次调用时必须指定的参数，而其他参数可由用户自行决定是否指定，如果已经有字典想作为自变量，那么也可以用 `ajax('http://openhome.cc', **my_settings)` 这样的方式，使用 `**` 进行拆解。

如果想要设计一个函数接受任意自变量，就可以在一个函数中同时使用 `*` 与 `**`，例如：

```
>>> def some(*arg1, **arg2):
...     print(arg1)
...     print(arg2)
...
>>> some(1, 2, 3)
(1, 2, 3)
{}
>>> some(a = 1, b = 22, c = 3)
()
{'a': 1, 'c': 3, 'b': 22}
>>> some(2, a = 1, b = 22, c = 3)
(2,)
{'a': 1, 'c': 3, 'b': 22}
>>>
```

4.2.3 一级函数的运用

在 Python 中，函数不单只是个定义，还是个值，定义的函数会产生一个函数对象，它是 `function` 的实例，既然函数是对象，也就可以赋值给其他的变量。例如：

```
>>> def max(num1, num2):
...     return num1 if num1 > num2 else num2
...
>>> maximum = max
>>> maximum(10, 5)
10
>>> type(max)
<class 'function'>
>>>
```

上面在定义了 `max()` 函数之后，通过 `max` 名称将函数对象赋值给 `maximum` 名称，无论通过 `max(10, 5)` 或者 `maximum(10, 5)`，结果都是调用了它们引用的函数对象。

函数与数值、列表、集合、字典、元组等一样，都被 Python 视为“一级公民”来对待，可以

自由地在变量、函数调用时指定（或赋值），因此具有这样特性的函数，也被称一级函数（First-class function），函数代表着某个可复用程序片段的封装，当它可以作为值传递时，就表示可以将某个可复用程序片段进行传递，它是极具威力的功能。

filter_lt()函数

如果有一个 `lt = ['Justin', 'caterpillar', 'openhomes']`，现在打算过滤出字符串长度大于 6 的元素，一开始可以编写如下的程序代码：

```
lt = ['Justin', 'caterpillar', 'openhomes']
result = []
for elem in lt:
    if len(elem) > 6:
        result.append(elem)
print(result)
```

如果需要多次进行这类的比较，那么可以定义出函数以复用这个程序片段。

```
def len_greater_than_6(lt):
    result = []
    for elem in lt:
        if len(elem) > 6:
            result.append(elem)
    return result

lt = ['Justin', 'caterpillar', 'openhomes']
print(len_greater_than_6(lt))
```

那么，如果想要过滤的长度小于 5 呢？在急着编写 `len_less_than_5()` 函数之前，先仔细想想，这类过滤某列表而后获得另一列表的程序片段，我们编写过几次了呢？每次其实只有过滤的条件不同，其他程序代码片段都是相同的，如果将重复的程序代码片段提取出来，封装为函数如何呢？



func filter_demo.py

```
def filter_lt(predicate, lt):
    result = []
    for elem in lt:
        if predicate(elem):
            result.append(elem)
    return result

def len_greater_than_6(elem):
    return len(elem) > 6

def len_less_than_5(elem):
    return len(elem) < 5

def has_i(elem):
    return 'i' in elem

lt = ['Justin', 'caterpillar', 'openhomes']
print('大于 6: ', filter_lt(len_greater_than_6, lt))
print('小于 5: ', filter_lt(len_less_than_5, lt))
print('有个 i: ', filter_lt(has_i, lt))
```

可以看到，将重复的程序代码片段提取出来后，就可以调用函数的方式代它们，然后每次给

予不同的函数来设置过滤条件。就目前来说，特别为 `len(elem) > 6`、`len(elem) < 5`、`'i' in elem` 使用 `def` 定义了 `len_greater_than_6()`、`len_less_than_5()`、`has_i()`，看起来有点小题大做，然而好处是只要看 `filter_lt(len_greater_than_6, lt)`、`filter_lt(len_less_than_5, lt)`、`filter_lt(has_i, lt)` 就可以很清楚地知道程序代码的目的。这个范例的执行结果如下：

```
大于 6: ['caterpillar', 'openhome']
小于 5: []
有个 i: ['Justin', 'caterpillar']
```

当然，读者可能觉得 `len_greater_than_6()` 不够通用，如果真是如此，那么也可以修改一下范例，让它更通用些。

```
func filter_demo2.py
```

```
def filter_lt(predicate, lt):
    result = []
    for elem in lt:
        if predicate(elem):
            result.append(elem)
    return result

def len_greater_than(num):
    def len_greater_than_num(elem):
        return len(elem) > num
    return len_greater_than_num

lt = ['Justin', 'caterpillar', 'openhome']
print('大于 5: ', filter_lt(len_greater_than(5), lt))
print('大于 7: ', filter_lt(len_greater_than(7), lt))
```

这次在 `len_greater_than()` 中定义了一个局部函数 `len_greater_than_num()`，之后将局部函数返回，返回的函数接受一个参数 `elem`，而本身带有调用 `len_greater_than()` 时传入的 `num` 参数值，因此 `len_greater_than(5)` 返回的函数相当于进行 `len(elem) > 5`，而 `len_greater_than(7)` 返回的函数相当于进行 `len(elem) > 7`，像这样调用函数返回（内部）另一个函数，也是函数作为一级公民的语言中常见的应用。

map_lt()函数

类似地，如果想将 `lt` 的元素全部转为大写后返回新的列表，一开始可能会直接编写以下的程序片段。

```
lt = ['Justin', 'caterpillar', 'openhome']
result = []
for ele in lt:
    result.append(ele.upper())
print(result)
```

同样地，将列表元素转换为另一组列表，也是曾经写过无数次的操作，何不将其中重复的程序片段抽取出来呢？

```
func map_demo.py
```

```
def map_lt mapper, lt):
```



```

result = []
for ele in lt:
    result.append(mapper(ele))
return result

lt = ['Justin', 'caterpillar', 'openhome']
print(map_lt(str.upper, lt))
print(map_lt(len, lt))

```

可以看到，将重复的程序片段提取出来后可以调用函数，然后每次给予不同的函数来设置对应转换的方式。当转换的函数定义好了，使用 `map_lt` 这样的函数就很方便，就像这里使用了 Python 标准链接库中的 `str.upper` 与 `len`。这个范例的执行结果如下：

```

>python map_demo.py
['JUSTIN', 'CATERPILLAR', 'OPENHOME']
[6, 11, 8]

```

filter()、map()、sorted()函数

实际上，Python 内建有 `filter()`、`map()` 函数可以直接调用它们。在 Python 3 中，`map()`、`filter()` 返回的实例并不是列表，分别是 `map` 与 `filter` 对象，都具有惰性求值的特性。下面来个简单的范例：

```
func filter_map_demo.py
```

```

def len_greater_than(num):
    def len_greater_than_num(elem):
        return len(elem) > num
    return len_greater_than_num

lt = ['Justin', 'caterpillar', 'openhome']
print(list(filter(len_greater_than(6), lt)))
print(list(map(len, lt)))

```

基本上，`filter()`、`map()` 能做得到的，for Comprehension 基本上都做得得到。在大多数情况下，for Comprehension 比较常见，不过有时通过适当的命名，使用 `filter()`、`map()` 会有比较好的可读性，像 `map(len, lt)` 就是一个例子。

再来看一个一级函数传递的例子，到目前为止，经常会使用列表、元组等有序结构，有时会将其中的元素进行排序，这时可以使用 `sorted()` 函数，它可以按照我们指定的方式进行排序。例如：

```

>>> sorted([2, 1, 3, 6, 5])
[1, 2, 3, 5, 6]
>>> sorted([2, 1, 3, 6, 5], reverse = True)
[6, 5, 3, 2, 1]
>>> sorted(('Justin', 'openhome', 'momor'), key = len)
['momor', 'Justin', 'openhome']
>>> sorted(('Justin', 'openhome', 'momor'), key = len, reverse = True)
['openhome', 'Justin', 'momor']
>>>

```

`sorted()` 会返回新的列表，其中包含了排序后的结果，`key` 参数可用来指定针对什么特性来迭代，例如在指定 `len()` 函数时，每个元素都会传入 `len()` 进行运算，得到的长度值再作为排序的依据。

如果是可变动的列表，本身也有个 `sort()` 方法，这个方法会直接在列表本身排序，不像 `sorted()`

方法会返回新的列表。例如：

```
>>> lt = [2, 1, 3, 6, 5]
>>> lt.sort()
>>> lt
[1, 2, 3, 5, 6]
>>> lt.sort(reverse = True)
>>> lt
[6, 5, 3, 2, 1]
>>> names = ["Justin", "openhome", "momor"]
>>> names.sort(key = len)
>>> names
['momor', 'Justin', 'openhome']
>>>
```

在 Python 标准链接库中，还有许多可接受函数值（或者返回函数）的函数，本书之后的章节也有机会看到一些应用。

4.2.4 lambda 表达式

在之前的 filter_demo.py 中，大费周章地为 `len(elem) > 6`、`len(elem) < 5`、`'i' in elem` 使用 `def` 定义了 `len_greater_than_6()`、`len_less_than_5()`、`has_i()`，它们的函数体其实都很简单，只有一句简单的运算，对于这类情况，可以考虑使用 `lambda` 表达式。例如：

```
func filter_demo3.py

def filter_lt(predicate, lt):
    result = []
    for elem in lt:
        if predicate(elem):
            result.append(elem)
    return result

lt = ['Justin', 'caterpillar', 'openhome']
print('大于 6: ', filter_lt(lambda elem: len(elem) > 6, lt))
print('小于 5: ', filter_lt(lambda elem: len(elem) < 5, lt))
print('有个 i: ', filter_lt(lambda elem: 'i' in elem, lt))
```

在 `lambda` 关键词之后定义的是参数，而冒号 (`:`) 之后定义的是函数体，运算的结果会作为返回值，不需要加上 `return`，像 `lambda elem: len(elem) > 6` 这样的 `lambda` 表达式会创建 `function` 实例，也就是一个函数。有时临时只是需要个小函数，使用 `lambda` 就很方便。

如果 `lambda` 不需要参数，直接在 `lambda` 后加上冒号就可以了，若需要两个以上的参数，中间要使用逗号 (,) 分隔开。例如：

```
>>> max = lambda n1, n2: n1 if n1 > n2 else n2
>>> max(10, 5)
10
>>>
```

在 Python 中缺少了其他语言中拥有的 `switch` 语句，有时会看到一些程序代码中结合字典与 `lambda` 来模拟 `switch` 的功能，姑且参考一下。



func grade.py

```
score = int(input('请输入分数: '))
level = score // 10
{
    10 : lambda: print('Perfect'),
    9 : lambda: print('A'),
    8 : lambda: print('B'),
    7 : lambda: print('C'),
    6 : lambda: print('D')
}.get(level, lambda: print('E'))()
```

在上例中，字典中的值是 lambda 创建的函数对象，程序中使用 get() 方法获取键对应的函数对象，如果键不存在，就返回 get() 第二个自变量指定的 lambda 函数，这是模拟了 switch 中 default 的部分。最后加上了 () 表示立即执行。

提示 >>>

和其他语言中的 lambda 语句相比，Python 使用 lambda 关键词的方式其实并不简洁，甚至有些妨碍可读性。实际上，Python 中的 lambda 也没办法写太复杂的逻辑，这是 Python 中为了避免 lambda 被滥用而特意做的限制。如果觉得可读性不佳，或者需要编写更复杂的逻辑，请乖乖使用 def 来定义函数，并给予一个清楚易懂的函数名称。

4.2.5 初探变量作用域

在 Python 中，变量无需事先声明，一个名称在赋值时，就可以成为变量，并创建起自己的作用域 (Scope)。在存取一个变量时，会看看当前作用域中是否有指定的变量名称，若无则向外寻找，因此在函数中可存取全局 (Global) 变量。

```
>>> x = 10
>>> def func():
...     print(x)
...
>>> func()
10
>>>
```

在上面的例子中，func() 中没有局部变量 x，因此往外寻找而获取了全局变量 x。如果在 func() 中对名称 x 作了赋值的操作呢？

```
>>> x = 10
>>> def func():
...     x = 20
...     print(x)
...
>>> func()
20
>>> print(x)
10
>>>
```

在 func() 中进行 x = 20 的时候，其实就创建了 func() 自己的局部变量 x，而不是将全局变量 x 设为 20，因此在 func() 执行完毕后，显示全局变量 x 的值仍会是 10。

就目前而言可以知道的是，变量可以在内建（**Builtin**）、全局（**Global**）、外包函数（**Endosing function**）、局部函数（**Local function**）中寻找或创建。范例如下：

```
func scope_demo.py

x = 10                # 创建全局 x

def outer():
    y = 20            # 创建全局 y

    def inner():
        z = 30        # 创建全局 z
        print('x = ', x) # 存取全局 x
        print('y = ', y) # 存取 outer() 函数的 y
        print('z = ', z) # 存取 inner() 函数的 z

    inner()

    print('x = ', x)    # 存取全局 x
    print('y = ', y)    # 存取 outer() 函数的 y

outer()
print('x = ', x)        # 存取全局 x
```

存取名称时（而不是对名称赋值）一定是从最内层往外寻找。**Python** 中的全局变量，实际上是以模块文件为界。以上例来说，**x** 实际上是 `scope_demo` 模块范围中的变量，作用域不会横跨所有模块。

我们经常使用的 `print` 名称属于内建作用域，在 **Python 3** 中有个 `builtins` 模块，该模块中的名称作用域横跨各个模块。例如：

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',
'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError',
'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError',
'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', ...略
```

`dir()` 函数可用来查询指定的对象上可存取的名称。在 **Python** 中可以直接使用的函数，其名称实际上是在 `builtins` 模块之中。基本上，也可以将变量创建到 `builtins`（但并不建议这么做）。例如：

```
import builtins
import sys
builtins.argv = sys.argv
print(argv[1])
```

在 **Python** 中有个 `locals()` 函数可用来查询局部变量的名称与值。例如：

```
func scope_demo2.py

x = 10

def outer():
    y = 20
```

```
def inner():
    z = 30
    print('inner locals:', locals())

inner()

print('outer locals:', locals())

outer()
```

执行的结果如下:

```
inner locals: {'z': 30}
outer locals: {'inner': <function outer.<locals>.inner at 0x01E11270>, 'y': 20}
```

在 Python 中还有个 `global()`，可以获取全局变量的名称与值，在全局作用域调用 `locals()` 时，获取的结果与 `global()` 是相同的。

如果对变量赋值时希望针对全局作用域，可以使用 `global` 来声明。例如：

```
>>> x = 10
>>> def func():
...     global x, y
...     x = 20
...     y = 30
...
>>> func()
>>> x
20
>>> y
30
>>>
```

来看看下面的代码会发生什么事情。

```
>>> x = 10
>>> def func():
...     print(x)
...     x = 20
...
>>> func()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in func
UnboundLocalError: local variable 'x' referenced before assignment
>>>
```

在 `func()` 函数中有个 `x = 20` 的赋值语句，python 解释器会认为 `print(x)` 中的 `x` 是 `func()` 函数中的局部变量 `x`，因为作用域创建总是在赋值时发生，就程序流程而言，在给 `x` 赋值之前就要显示出其值就是错误的。如果是想显示全局的 `x` 值，那么在 `print(x)` 前行使用 `global x` 声明，就可以避免这个问题。

当然，无论是哪种程序设计语言，除非那些概念上真的是全局的名称，否则都不鼓励使用全局变量，因此在 Python 中应避免 `global` 声明的使用。

在 Python 3 中新增了 `nonlocal`，可以指明变量并非局部变量，请解释器按照局部函数、外包函数、全局、内建的顺序来寻找变量，就算是赋值运算时，也要求是这个顺序。例如：

func scope_demo3.py

```

x = 10
def outer():
    x = 100    # 这是在 outer() 函数作用域的 x
    def inner():
        nonlocal x
        x = 1000    # 改变的是 outer() 函数的 x
    inner()
    print(x)    # 显示 1000

outer()
print(x)    # 显示 10

```

变量作用域的讨论虽然略显无趣，然而若没有搞清楚相关规则，则很容易发生名称冲突，导致一些不可预期的程序错误，不可不慎。目前暂时先对一个模块文件中相关的作用域进行探讨，之后有机会还会探讨其他有关作用域的议题。

4.2.6 yield 与 yield from

我们可以在函数中使用 `yield` 来产生值，表面上看来，`yield` 有点像是 `return`，不过函数并不会因为 `yield` 而结束，只是将流程控制权转交给函数的调用者。下面来模仿 `range()` 函数的实现，自定义一个 `xrange()` 函数。

func yield_demo.py

```

def xrange(n):
    x = 0
    while x != n:
        yield x
        x += 1

for n in xrange(10):
    print(n)

```

就流程来看，`xrange()` 函数首次执行时，使用 `yield` 产生一个值，然后回到主流程使用 `print()` 显示该值，接着流程重回 `xrange()` 函数 `yield` 之后继续执行，循环中再度使用 `yield` 赋值，然后又回到主流程使用 `print()` 显示该值，这样的反复流程会直到 `xrange()` 中的 `while` 循环结束为止。

显然，这样的流程有别于函数中使用 `return` 函数就结束了的情况。实际上，当函数中使用 `yield` 产生一个值时，调用该函数会返回一个 `generator` 对象，也就是一个生成器，此对象具有 `__next__()` 方法，通常会使用 `next()` 函数调用该方法取出下一个产生值（也就是 `yield` 指定的值），若无法产生下一个值（也就是含有 `yield` 的函数结束了），则会发生 `StopIteration` 例外（Exception）。

```

>>> g = xrange(2)
>>> type(g)
<class 'generator'>
>>> next(g)
0
>>> next(g)
1
>>> next(g)

```



```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

因此 for in 实际上是对 xrange() 返回的生成器进行迭代, 它会调用 __next__() 方法获取 yield 产生的值, 并在遇到 StopIteration 时结束迭代。因为每次调用生成器的 __next__() 时, 生成器才会运算并返回下一个产生值, 因此就解释了先前为何提到生成器, 它具有惰性求值的效果。

提示 >>>

在 4.1.5 小节讨论 for Comprehension 时曾说过, 可以使用 () 括住 for Comprehension, 这样会创建一个 generator 对象, 这个对象也可以使用 for in 来迭代。

yield 实际上是个表达式, 除了可以调用生成器的 __next__() 方法, 获取 yield 的右侧产生值之外, 还可以通过 send() 方法来产生值, 令其成为 yield 运算的结果, 也就是生成器可以给调用者产生值, 调用者也可以赋值给生成器, 这成了一种沟通机制。例如设计一个简单的生产者与消费者程序。

func producer_consumer.py

```
import sys
import random

def producer():
    while True:
        data = random.randint(0, 9)
        print('生产了: ', data)
        yield data  ← ❶ 产生下一个值, 流程回到调用者

def consumer():
    while True:
        data = yield  ← ❷ 调用产生器 send() 方法时的产生值
                        会成为 yield 的运算结果
        print('消费了: ', data)

def clerk(jobs, producer, consumer):
    print('执行 {} 次生产与消费'.format(jobs))
    p = producer()
    c = consumer()
    next(c)  ← ❸ 令消费者执行至 yield 处
    for i in range(jobs):
        data = next(p)  ← ❹ 获取生产者的产生值
        c.send(data)  ← ❺ 将值传给消费者

clerk(int(sys.argv[1]), producer, consumer)
```

由于 send() 方法是 yield 的运算结果, 因此 clerk() 流程中必须先使用 next(c) ❸, 使得流程首次执行到 consumer() 函数中 data = yield 处时先执行 yield ❷, 执行 yield 会令流程回到 clerk() 函数, 之后执行至 next(p) ❹, 这时流程进行到 producer() 函数的 yield data ❶, 在 clerk() 获取 data 之后, 接着执行 c.send(data) ❺, 这时流程回到 consumer() 之前 data = yield 处, send() 方法的产生值此时成为 yield 的结果。执行结果如下:

```
>python producer_consumer.py 3
执行 3 次生产与消费
生产了: 4
消费了: 4
生产了: 5
消费了: 5
生产了: 9
消费了: 9
```

提示 >>>

如果想对生成器引发例外，可以使用 `throw()` 方法，这个方法接受的三个自变量为例外类型、实例以及 `traceback` 对象，有关例外处理将在第 7 章加以说明。

`yield` 可用来创建生成器。在 Python 中，也有许多函数的执行结果是返回生成器，如果善加利用，就可以提高程序的执行效率。如果打算创建一个生成器函数，然而数据源是直接从另一个生成器获得，那会怎么样呢？举例来说，`range()` 函数就是返回生成器，而我们打算创建一个 `np_range()` 函数产生指定数字的正负范围，但不包含 0。

```
def np_range(n):
    for i in range(0 - n, 0):
        yield i

    for i in range(1, n + 1):
        yield i
# 显示[-5, -4, -3, -2, -1, 1, 2, 3, 4, 5]
print(list(np_range(10)))
```

因为 `np_range()` 必须是个生成器，结果就是要逐一从源生成器获取数据，再将之通过 `yield` “产生”，像这里重复使用了 `for in` 来迭代。从 Python 3.4 开始，新增了 `yield from` 语句，上面的程序片段可以直接改写为以下方式：

```
def np_range(n):
    yield from range(0 - n, 0)
    yield from range(1, n + 1)
# 显示[-5, -4, -3, -2, -1, 1, 2, 3, 4, 5]
print(list(np_range(10)))
```

当需要直接从某个生成器获取数据，以便创建另一个生成器时，`yield from` 可以作为直接衔接的语句。

提示 >>>

实际上，`yield from` 语句本来是要用来与 Python 的 `asyncio` 模块搭配，实现基于生成器的异步功能，然而语义不够清楚，因此在 Python 3.5 中增加了 `async`、`await` 加以替代，`yield from` 基于兼容性而暂时保留着，因而我们不建议再使用。然而，读者可能会在某些地方看到相关程序代码或文件含有这条语句，所以在这里基于本书介绍内容的完整性而进行了说明。

`asyncio` 是个很大的主题，本书并没有涉及，若读者感兴趣，可参考“`Asynchronous I/O, event loop`”，

coroutines and tasks¹”。

4.3 重点复习

在 Python 中，程序区块使用冒号 (:) 开头，之后同一区块范围要有相同的缩排，不可混用不同空格的数量，不可混用空格与 Tab，Python 的建议是使用 4 个空格进行缩排。

在 Python 中有一个 if...else 表达式语句，当 if 的条件判断式成立时，会返回 if 前的数值，若不成立则返回 else 后的数值。Python 提供了 while 循环，可根据指定条件判断式来判断是否执行循环体。如果想要按序迭代某个序列，例如字符串、列表、元组，那么可以使用 for in 语句。

range()函数的形式是 range(start, stop[, step])，start 省略时默认是 0，step 是步进值，省略时默认是 1。可以使用 zip()函数将两个序列的各个元素像拉链般一对一配对结合起来，实际上 zip()可以接受多个序列。如果真的要迭代时具有索引信息，那么使用 enumerate()函数可能是最方便的。

只要实现了__iter__()方法的对象，都可以使用 for in 来迭代；只要实现了__iter__()方法的对象，都可以通过__iter__()方法返回一个迭代器，这个迭代器可以使用 for in 来迭代。

集合也实现了__iter__()方法，因此可以进行迭代，想要迭代字典键与值，可以使用它的 keys()、values()或 items()方法，它们分别会返回 dict_keys、dict_values、dict_items 对象，它们都实现了__iter__()方法，因此也可以使用 for in 迭代。

有时在某个区块中并不想执行任何程序语句，或者稍后才会编写些什么语句，对于还没打算编写任何程序语句的区块，可以先放个 pass。

break 可分别用来中断 while 循环、for in 的迭代。在 while 循环中遇到 continue，不执行此次循环后续的程序代码，直接进行下一次循环。在 for in 迭代遇到 continue，不执行此次迭代后续的程序代码，直接进行下一次迭代。

将一个列表转换为另一个列表，是很常见的操作，Python 对这类需求提供了 for Comprehension 语句。for Comprehension 也可以与条件判断式结合，这可以构成一个过滤的功能。

在 for Comprehension 两旁放上[]表示会产生列表，如果数据源很长或者数据源本身是个有惰性求值特性的生成器时，直接产生列表显得没有效率，这时可以在 for Comprehension 两旁放上()，这样就会创建一个 generator 对象，具有惰性求值特性。

for Comprehension 也可以用来创建集合，只要在 for Comprehension 两旁放上{}。使用 for Comprehension 来创建字典实例也是可行的。想要用 for Comprehension 创建元组 (tuple)，可以将 for Comprehension 生成器表达式传给 tuple()。

如果函数执行完毕却没有使用 return 返回值，或者使用了 return 结束函数却没有指定返回值，默认就会返回 None。

在 Python 中，函数中还可以定义函数，称为局部函数 (Local function)。在 Python 中可以使用默认自变量、关键词自变量。

若有个函数拥有固定的参数，而我们有一个序列，像列表、元组，只要在传入时加上*，则列

¹ Asynchronous I/O, event loop, coroutines and tasks: docs.python.org/3/library/asyncio.html

表或元组中各个元素就会自动拆解给各个参数。可以在定义函数的参数时使用`*`，表示该参数接受不定长度的自变量。

如果有一个字典，打算按照键名称赋值给对应的参数名称，可以在字典前加上`**`，这样字典中的各对键和值就会自动拆解给各个参数。可以试着使用`**`来定义参数，让指定的关键词自变量收集为一个字典。

如果想要设计一个函数接受任意自变量，可以在一个函数中同时使用`*`与`**`。

在 Python 中，函数不单只是个定义，还是个值，定义的函数会产生一个函数对象，它是 `function` 的实例，既然函数是对象，也就可以赋值给其他的变量。

要将列表中的元素进行排序，可以使用 `sorted()` 函数。如果是可变动的列表，本身也有个 `sort()` 方法，这个方法会直接在列表本身排序。

如果函数体很简单，只有一句简单的运算，那么可以考虑使用 `lambda` 运算式。

一个名称在赋值时就可以成为变量，并创建起自己的作用域，在存取一个变量时，会看看当前作用域中是否有指定的变量名称，若无则向外寻找。

变量可以在内建（Builtin）、全局（Global）、外包函数（Endosing function）、局部函数（Local function）中寻找或创建。Python 中的全局作用域，实际上是以模块文件为界。

`dir()` 函数可用来查询指定的对象上可存取的名称。Python 中可以直接使用的函数，其名称实际上是在 `builtins` 模块之中。在 Python 中有个 `locals()` 函数，可用来查询局部变量的名称与值。Python 中还有个 `global()`，可以获取全局变量的名称与值，当我们在全局作用域调用 `locals()` 时，获取的结果与 `global()` 是相同的。

如果对变量赋值时希望针对全局作用域，可以使用 `global` 来声明。在 Python 3 中新增了 `nonlocal`，可以指明变量并非局部变量，请解释器按照局部函数、外包函数、全局、内建的顺序来寻找变量，就算是赋值运算时也要求是这个顺序。

可以在函数中使用 `yield` 来产生值，表面上看来，`yield` 有点像是 `return`，不过函数并不会因为 `yield` 而结束，只是将流程控制权转交给函数的调用者。`yield` 实际上是个表达式，除了调用生成器的 `__next__()` 方法，获取 `yield` 的右侧产生值之外，还可以通过 `send()` 方法产生值，令其成为 `yield` 运算的结果。

课后练习

实践题

1. 在三位的整数中，153 可以满足 $1^3 + 5^3 + 3^3 = 153$ ，这样的数称之为阿姆斯特朗（Armstrong）数，试用程序找出所有三位数的阿姆斯特朗数。

2. Fibonacci（斐波拉契）为十三世纪（1200 年代）欧洲的数学家，在他的著作中提过，若一只兔子每月生一只小兔子，一个月后小兔子也开始生产。起初只有一只兔子，一个月后有两只兔子，二个月后有两只兔子，三个月后有五只兔子……也就是每个月兔子总数会是 1、1、2、3、5、8、13、21、34、55、89……这就是斐波拉契数列，可用公式定义如下：

```
fn = fn-1 + fn-2 if n > 1
fn = n if n = 0, 1
```

请编写程序，可让用户输入想计算的斐波拉契数列的个数，再由程序全部显示出来。例如：

```
求几个斐波拉契数列的个数？10
0 1 1 2 3 5 8 13 21 34
```

3. 请编写一个简单的洗牌程序，可在文本模式下显示洗牌结果。例如：

```
桃9 心10 梅4 桃J 砖5 梅10 梅K 砖9 梅J 砖2 砖A 心6 心5
桃8 梅2 砖6 梅3 梅7 梅A 心4 心J 心8 心Q 梅6 砖J 心K
桃6 砖8 心7 桃5 砖K 砖3 心A 桃7 梅9 心9 桃3 砖10 心3
桃A 桃4 桃2 桃10 桃Q 砖7 梅8 心2 梅Q 梅5 砖Q 桃K 砖4
```

4. 试着使用 for Comprehension 来找出周长为 24，每个边长都为整数且不超过 10 的直角三角形的边长。

第 5 章

从模块到类

学习目标

- 深入模块管理
- 初识面向对象
- 学习定义类
- 定义运算符



5.1 模块管理

Python 是个支持多范式的程序设计语言，无论采取过程式、函数式或面向对象，在构建程序时都应该思考以下几个重点。

- 抽象层的封装与隔离。
- 对象的状态。
- 命名空间。
- 资源实体的组织方式，像源码文件、软件包等。

在第 4.2 节讨论过 Python 中如何定义函数，函数是一个抽象层，用来封装算法的流程细节。对于函数的调用者而言，最好的方式是了解函数的接口，也就是仅需知道函数名称、参数、返回值这样的“外观”，而不用知道函数的实现细节。

在 Python 中，模块也提供了一种抽象层的封装与隔离，2.2 小节曾简单介绍过模块，这一节要来深入模块的细节，了解如何善用模块来建立最自然的抽象层。

5.1.1 用模块建立抽象层

如同 2.2 小节中谈到的，一个.py 文件就是一个模块，这使得模块成为 Python 中最自然的抽象层。

就算一开始只会创建.py 源码文件，在当中定义一些基本的变量，读者也可以试着将变量分门别类，放在不同名称的.py 文件中，像是将一些数学相关的常数（如圆周率 π 、自然对数 e ）放在 `xmath.py` 文件之中。这么一来，在想要使用这些数学相关常数时，就能 `import xmath`，之后以 `xmath.pi`、`xmath.e` 的方式来引用，而无需再重复编写相关常数。

实际上 Python 就内建有 `math` 模块，其中除了定义圆周率 π 、自然对数 e 之外，还有一些常用的数学函数的定义，像三角函数、`log()`、`pow()` 等。想要知道一个模块中有哪些名称，可以使用 `dir()` 函数。例如：

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> dir(math)
['_doc_', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf',
'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd',
'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'trunc']
>>>
```

这就是模块作为一种抽象层的封装与隔离的好处，我们可以用一个模块名称来组织或思考一个整体功能。实际上，当 `import` 某个模块而使得指定的.py 文件被加载时，python 解释器会为它

创建一个 **module** 实例，并创建一个模块名称来引用它，`dir(math)` 实际上是查询 `math` 名称引用的 **module** 实例上有哪些属性（Attribute）名称可以存取。

提示 >>>

若调用 `dir()` 时未指定任何 **module** 实例，则会查询当前所在模块的 **module** 实例上拥有的名称。

本节的 `math` 模块就是个例子，在 4.2 小节学会了如何将可复用程序片段定义为函数之后，若想要设计一个银行业务相关的简单程序，让它具有创建账户、存款、提款等功能，何不将这些函数定义在一个 `bank.py` 中呢？

modules bank.py

```
def account(name, number, balance):
    return {'name': name, 'number': number, 'balance': balance}

def deposit(acct, amount):
    if amount <= 0:
        print('存款金额不得为负')
    else:
        acct['balance'] += amount

def withdraw(acct, amount):
    if amount > acct['balance']:
        print('余额不足')
    else:
        acct['balance'] -= amount

def desc(acct):
    return 'Account:' + str(acct)
```

接下来在其他的 `.py` 文件中，只要 `import bank` 就可以通过 `bank` 模块名称来进行相关的业务流程了。例如：

modules bank_demo.py

```
import bank

acct = bank.account('Justin', '123-4567', 1000)
bank.deposit(acct, 500)
bank.withdraw(acct, 200)

# 显示 Account: {'balance': 1300, 'number': '123-4567', 'name': 'Justin'}
print(bank.desc(acct))
```

通过 `bank.account()`、`bank.deposit()`、`bank.withdraw()`、`bank.desc()` 这样的名称可以很清楚地看到账户的创建、存款、提款、描述都是与银行业务相关的操作，`bank` 这个名称不单只是用来避免命名空间的冲突，也是作为一种组织与思考相关功能的方式。

就目前为止，这个简单程序只使用了模块来管理创建账户、存款、提款等函数，然而这些函数都与一个记录账户状态的字典对象相关，稍后还会看到更好的方式来组织函数与对象的状态。

5.1.2 管理模块名称

大部分情况下模块都会被拿来作为命名空间。之前介绍过,当 `import` 某个模块而使得指定的.py 文件被加载时,python 解释器会为它创建一个 `module` 实例,并创建一个模块名称来引用它,这是最简单的情况。然而,Python 中管理模块名称的方式还有着其他的可能性。

from import 名称管理

在 4.2 小节说过,可以使用 `from import` 在当前模块创建和被导入模块相同的名称。事实上,`from import` 会将被导入模块中名称引用的值赋给当前模块中创建的新名称。例如,在 `foo.py` 文件中定义了一个 `x` 变量:

```
x = 10
```

若在另一个 `main.py` 文件中执行 `from foo import x`,实际上会在 `main` 模块中创建一个 `x` 变量,然后将 `foo` 中 `x` 的值 10 赋值给 `main` 中的 `x` 变量,因此会产生以下的结果:

```
>>> from foo import x
>>> x
10
>>> x = 20
>>> import foo
>>> foo.x
10
>>>
```

简单来说,当执行 `from foo import x` 时,就是在模块中创建了新变量,而不是使用原本的 `foo.x` 变量,只是一开始两个变量引用同一个值。若引用了可变动对象,就要特别小心了。例如,若 `foo.py` 中编写了:

```
lt = [10, 20]
```

就会产生以下的结果:

```
>>> from foo import lt
>>> lt
[10, 20]
>>> lt[0] = 15
>>> import foo
>>> foo.lt
[15, 20]
>>>
```

这是因为 `lt` 变量与 `foo.lt` 都引用了同一个列表 (list) 对象,因此通过 `lt` 变量修改索引 0 的元素使用 `foo.lt` 就会获取修改后的结果。

限制 from import *

使用 `from import` 语句时,若最后是 `*` 结尾,则会将被导入模块中所有的变量,在当前模块中创建相同的名称。如果有些变量并不想被 `from import *` 创建同名变量,可以用下划线作为开头。例如,若 `foo.py` 中有以下内容:

```
x = 10
lt = [10, 20]
_y = 20
```


使用 `from foo import *` 时，当前模块中并不会创建 `_y` 变量。例如：

```
>>> from foo import *
>>> x
10
>>> lt
[10, 20]
>>> _y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_y' is not defined
>>>
```

想避免 `from import *` 被滥用而“污染”了命名空间时，就可以使用这种方式。另一个方式是定义一个 `__all__` 列表，使用字符串列出可被 `from import *` 的名称，例如：

```
__all__ = ['x', 'lt']

x = 10
lt = [10, 20]
_y = 20
z = 30
```

如果模块中定义了 `__all__` 变量，那么就只有名单中的变量才可以被其他模块 `from import *`。例如：

```
>>> from foo import *
>>> x
10
>>> lt
[10, 20]
>>> _y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_y' is not defined
>>> z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
>>>
```

无论是下划线开头，还是未被列入 `__all__` 列表的名称，只是限制不被 `from import *`，如果用户 `import foo`，那么依旧可以使用 `foo._y` 或 `foo.z` 来存取。

```
>>> import foo
>>> foo._y
20
>>> foo.z
30
>>>
```

del 模块名称

在 3.2.1 小节讨论变量时曾经提过 `del`，它可以将已创建的变量删除。被 `import` 的模块名称或者 `from import` 创建的名称实际上就是个变量，因此可以使用 `del` 将模块名称或者 `from import` 的名称删除。例如：

```
>>> import foo
>>> foo.x
10
>>> del foo
>>> foo.x
```

```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'foo' is not defined
>>> from foo import x
>>> x
10
>>> del x
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>

```

因此，如果想模拟 `import foo as qoo`，那么可以如下来实现。例如：

```

>>> import foo
>>> qoo = foo
>>> del foo
>>> qoo.x
10
>>> foo.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'foo' is not defined
>>>

```

sys.modules

`del` 用来删除指定的名称而不是删除名称引用的对象本身，举例来说：

```

>>> lt1 = [1, 2]
>>> lt2 = lt1
>>> del lt1
>>> lt1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'lt1' is not defined
>>> lt2
[1, 2]
>>>

```

在上例中，虽然执行了 `del lt1`，然而 `lt2` 还是引用着列表实例。同样的道理，`del` 时若指定了模块名称，只是将该名称删除，而不是删除 `module` 实例。实际上，要知道当前已加载的 `module` 名称与实例有哪些，可以通过 `sys.modules`，这是个字典对象，键的部分是模块名称，值的部分是 `module` 实例。例如：

```

>>> import sys
>>> import foo
>>> 'foo' in sys.modules
True
>>> foo.x
10
>>> del foo
>>> foo.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'foo' is not defined
>>> sys.modules['foo'].x
10
>>>

```

在上面的例子中可以看到, `del foo` 删除了 `foo` 名称, 然而, 我们还是可以通过 `sys.modules['foo']` 存取到 `foo` 原来引用的 `module` 实例。

模块名称的作用域

到目前为止, 都是在全局作用域使用 `import`、`import as`、`from import`, 实际上它们也可以出现在语句能出现的位置, 例如 `if...else` 区块或者函数之中, 因此能根据不同的情况进行不同的 `import`。

当使用 `import`、`import as`、`from import` 时, 创建的名称其实就是变量名称, 因此根据使用 `import`、`import as`、`from import` 的位置所创建的名称也会有其作用域。例如, 在函数中使用了 `import foo`, 那么 `foo` 这个名称的作用域就只会在函数之中。

```
>>> def qoo():
...     import foo
...     print(foo.x)
...
>>> qoo()
10
>>> foo.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'foo' is not defined
>>>
```

5.1.3 设置 PTH 文件

在 2.2.2 小节介绍过 `sys.path`, 这个列表中列出了寻找模块时的路径, 列表内容基本上来自以下几个来源。

- 执行 `python` 解释器时的文件夹。
- `PYTHONPATH` 环境变量。
- `Python` 安装的标准链接库等文件夹。
- `PTH` 文件列出的文件夹。

在 2.2.2 小节中, 基本上已经说明过前三个来源, 介绍了针对 `sys.path` 增删路径, 可以影响模块的搜索路径。至于 `PTH` 文件的部分, 就是指可以在一个 `.pth` 文件中列出模块的搜索路径, 一行一个路径。例如:

modules workspace.pth

```
C:\workspace\libs
C:\workspace\third-party
C:\workspace\devs
```

`PTH` 文件的存放位置不同操作系统并不相同, 可以通过 `site` 模块的 `getsitepackages()` 函数获取正确的位置, 以 Windows 的 `Python` 安装版本为例, 会显示以下的位置:

```
>>> import site
>>> site.getsitepackages()
['C:\\Program Files (x86)\\Python35-32', 'C:\\Program Files (x86)\\Python35-32\\lib\\site-packages']
>>>
```


如果确实创建了 **workspace.pth** 中列出的文件夹, 而且将 **workspace.pth** 放置到 **C:\Program Files (x86)\Python35-32** 中, 那么 **sys.path** 就会是以下的结果:

```
>>> import sys
>>> sys.path
['', 'C:\\Program Files (x86)\\Python35-32\\python35.zip', 'C:\\Program Files (x86)\\Python35-32\\DLLs', 'C:\\Program Files (x86)\\Python35-32\\lib', 'C:\\Program Files (x86)\\Python35-32', 'C:\\workspace\\libs', 'C:\\workspace\\third-party', 'C:\\workspace\\devs', 'C:\\Program Files (x86)\\Python35-32\\lib\\site-packages']
>>>
```

注意 >>>

如果仅在 PTH 文件中列出路径, 却没有创建对应的文件夹, 那么 **sys.path** 中并不会加入那些路径。

如果将 **workspace.pth** 放置到 **C:\Program Files (x86)\Python35-32\lib\site-packages** 中, 那么 **sys.path** 就会是以下的结果:

```
>>> import sys
>>> sys.path
['', 'C:\\Program Files (x86)\\Python35-32\\python35.zip', 'C:\\Program Files (x86)\\Python35-32\\DLLs', 'C:\\Program Files (x86)\\Python35-32\\lib', 'C:\\Program Files (x86)\\Python35-32', 'C:\\Program Files (x86)\\Python35-32\\lib\\site-packages', 'C:\\workspace\\libs', 'C:\\workspace\\third-party', 'C:\\workspace\\devs']
>>>
```

如果想将 PTH 文件放置到其他文件夹, 可以使用 **site.addsitedir()** 函数新增 PTH 文件的文件夹来源, 例如可以将 **workspace.pth** 放置到 **C:\workspace** 中, 接着如下操作:

```
>>> import site
>>> site.addsitedir('C:\\workspace')
>>> import sys
>>> sys.path
['', 'C:\\Program Files (x86)\\Python35-32\\python35.zip', 'C:\\Program Files (x86)\\Python35-32\\DLLs', 'C:\\Program Files (x86)\\Python35-32\\lib', 'C:\\Program Files (x86)\\Python35-32', 'C:\\Program Files (x86)\\Python35-32\\lib\\site-packages', 'C:\\workspace', 'C:\\workspace\\libs', 'C:\\workspace\\third-party', 'C:\\workspace\\devs']
>>>
```

除了 **workspace.pth** 中列出的路径之外, **site.addsitedir()** 函数新增的路径也会是 **sys.path** 路径中的一部分。

提示 >>>

若有兴趣了解细节, 下面列出了一个模块被 **import** 时发生了的事情。

1. 在 **sys.path** 寻找模块。
2. 加载、编译模块的程序代码。
3. 创建空的模块对象。
4. 在 **sys.modules** 中记录该模块。
5. 执行模块中的程序代码及相关定义。

5.2 初识面向对象

函数是个抽象层，封装了算法的流程细节。模块是个抽象层，可以用模块名称来组织或思考一个整体功能，那么 Python 中类应用的场合呢？Python 中不是一切都是对象吗？什么时候该以对象来思考或组织应用程序的行为呢？这可以从打算将对象的状态与功能“粘”在一起时开始。

5.2.1 定义类

在 5.1.1 小节中曾经创建了一个 bank.py，其中是有关账户创建、存款、提款等的函数。实际上，bank 模块中的函数操作都是与传入的字典实例（即代表账户状态的对象）高度相关的，何不将它们组织在一起呢？我们可以为账户创建一个专用类，拥有专用属性，然后让存款、提款等函数专属于这个账户类的实例，这样在设置对象状态、思考对象可用的操作时都会比较方便一些。

我们直接来修改 bank.py 中的程序代码，看看是否真的能增加可用性（Usability）。在 Python 中可以使用 class 来创建一个专用类，bank.py 第一次修改后的成果如下。

object-oriented1 bank.py

```
class Account:  ← ❶ 定义 Account 类
    pass
```

```
def account(name, number, balance):
```

```
    acct = Account()
    acct.name = name
    acct.number = number
    acct.balance = balance
    return acct
```

← ❷ 创建 Account 实例并设置相关属性

```
def deposit(acct, amount):
```

```
    if amount <= 0:
        print('存款金额不得为负')
    else:
```

```
        acct.balance += amount ← ❸ 使用点运算符（.）存取相关属性
```

```
def withdraw(acct, amount):
```

```
    if amount > acct.balance:
        print('余额不足')
    else:
        acct.balance -= amount
```

```
def desc(acct):
```

```
    return "Account('{name}', '{number}', {balance})".format(
        name = acct.name, number = acct.number, balance = acct.balance
    )
```

在这个范例中定义了 Account 类❶，类是对象的蓝图，目前还没有在蓝图中加上任何定义，只是单纯地放置了 pass，我们的目的只是先让账户有个专用类 Account。

想要创建 Account 实例，可以调用 Account()（这相当于按照蓝图来制作出一个成品），接着在实例上设置相关属性❷，这么一来，类与属性就有了特定关联，也就是想到的 Account，还有 name、

number、balance 等, 进一步可在 Account 上使用点运算符 (.) 来存取这些属性⑨, 这会比原先使用字典实例要好, 毕竟键和值的存取操作专属于字典这个类型。

由于 account()、deposit()、withdraw()、desc() 函数的外观并没有改变, 只是更改了内部实现, 因此完成了 bank.py 的修改后可以直接执行 bank_demo.py 程序。

5.2.2 定义方法

虽然我们定义了 Account 类作为账户的专用类, 然而 account()、deposit()、withdraw()、desc() 函数却是在其他地方定义, 明明它们都是与 Account 实例相关的操作, 将相关的操作放在一起而不是分开是设计时的一个基本原则, 面向对象更是如此。

⑨ 定义 __init__() 方法

来看看 account() 函数, 它定义了如何创建实例, 以及实例创建后的相关属性设置, 这是每个 Account 实例都要经历的初始化流程, 可以将初始化流程使用 __init__() 方法定义在类之中。例如:

```
class Account:
    def __init__(self, name, number, balance):
        self.name = name
        self.number = number
        self.balance = balance
```

记得先将 pass 删除, 然后定义 __init__() 方法接着就可以将原先的 account() 函数删除。

可以看到方法前后各有两个连接的下划线, 在 Python 中, 这样的名称意味着在类以外的其他位置不要直接调用, 基本上都会有个函数来调用这类方法, 就 __init__() 而言, 如果如下创建 Account 实例时就会调用这个初始化方法。

```
acct = Account('Justin', '123-4567', 1000)
```

在调用 __init__() 方法时, 创建的 Account 实例会传入作为方法的第一个参数, 虽然第一个参数的名称可以自定义, 然而在 Python 的惯例中, 第一个参数的名称会命名为 self。

在创建类实例时, 如果有其他的参数, 可以从第二个参数开始按序定义, 若要设置实例属性, 可以通过 self 与点运算符来设置, __init__() 方法不需使用 return self 来将创建的实例返回, acct = Account('Justin', '123-4567', 1000) 执行过后, 就会将创建的实例赋值给 acct。

⑨ 定义操作方法

接着来将 deposit() 以及 withdraw() 也定义在 Account 类之中。例如:

```
class Account:
    先前的 __init__() 定义...故略

    def deposit(self, amount):
        if amount <= 0:
            print('存款金额不得为负')
        else:
            self.balance += amount

    def withdraw(self, amount):
        if amount > self.balance:
```



```

        print('余额不足')
    else:
        self.balance -= amount

```

在将 deposit()、withdraw() 移到 Account 类中之后，主要的修改在于第一个参数名称，在 Python 中，对象的方法的第一个参数一定是对象本身，就 Python 的设计哲学（Zen of Python）来说，这是“Explicit is better than implicit”（显式比隐式更好）的实践。因为在方法的实现中，以 self 作为前置的名称一定就是对实例属性进行存取，而不会是针对局部变量。

稍后也会看到，定义在 Account 中的 __init__()、deposit()、withdraw() 本质上也是函数。不过在面向对象的术语中，对这些定义在类中可对对象进行的操作，习惯上被称为方法（Method）。

定义 __str__() 方法

目前有个 desc() 函数，还没定义到 Account 类之中，虽然可以如 deposit()、withdraw() 那样直接将 desc() 定义在 Account 中，然后将第一个参数更名为 self，不过像 desc() 这个会返回对象描述字符串的方法，在 Python 中有个特殊名称 __str__() 专门用来定义这个操作。

```

class Account:
    之前定义过的方法...故略

    def __str__(self):
        return "Account('{name}', '{number}', {balance})".format(
            name = self.name, number = self.number, balance = self.balance
        )

```

同样地，方法的第一个参数定义为 self，用来接受对象本身，正如前文所述，方法前后各有两个连接下划线，在 Python 中意味着不要直接去调用，基本上会有函数来调用这类方法，就 __str__() 而言，print() 方法是个实例，在执行 print(acct) 这样的语句时就会调用 acct 的 __str__() 方法获取描述字符串，然后再使用 print() 来显示。

另一个例子是 str()，如果执行 str(acct)，就会调用 acct 的 __str__() 方法获取描述字符串并返回，这时可以回忆一下，3.2.2 小节曾经稍微提过 str() 与 repr()，实际上类中也可以定义 __repr__() 方法，当执行 repr(acct) 时，就会调用 acct 的 __repr__() 方法获取描述字符串并返回。

虽然 __str__() 与 __repr__() 返回的字符串描述也可以相同，不过 __str__() 字符串描述主要是给人类看的易懂格式，而 __repr__() 是给程序、计算机解析用的特定格式（像对代表日期的字符串进行解析，以便创建一个日期对象），或者是包含调试用的字符串信息。

提示 >>>

Python 内建类的 __repr__() 返回的字符串是个有效的 Python 表达式（expression），可以使用 eval() 运算来产生一个内含值相同的对象。

从 __init__()、__str__()、__repr__() 的例子中可知，在 Python 中，__xxx__() 这类有着特定意义的方法名称还真不少，之后还会看到更多。

为了便于了解全部修改后的结果，来看看现在的 bank.py 内容是什么样子。

```

object-oriented2 bank.py

```

```

class Account:

```

```

def __init__(self, name, number, balance):
    self.name = name
    self.number = number
    self.balance = balance

def deposit(self, amount):
    if amount <= 0:
        print('存款金额不得为负')
    else:
        self.balance += amount

def withdraw(self, amount):
    if amount > self.balance:
        print('余额不足')
    else:
        self.balance -= amount

def __str__(self):
    return "Account('{name}', '{number}', {balance})".format(
        name = self.name, number = self.number, balance = self.balance
    )

```

这样的修改主要是为了便于客户端的使用，由于调用方式改变了，因此 `bank_demo.py` 也必须进行修改。



object-oriented2 bank_demo.py

```

import bank

acct = bank.Account('Justin', '123-4567', 1000)
acct.deposit(500)
acct.withdraw(200)

# 显示 Account('Justin', '123-4567', 1300)
print(acct)

```

可以看到，需要进行存款、提款操作时，使用的是专属于 `Account` 的 `deposit()`、`withdraw()` 方法，这样就容易使用多了。如果 IDE 支持智能提示，还能自动出现对象上可用的操作进行选取，这样就更方便了，如图 5-1 所示。

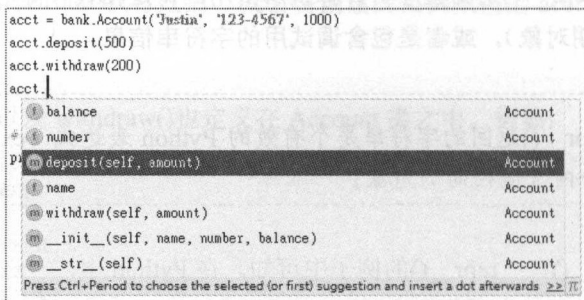


图 5-1 IDE 的智能提示

提示>>>

是的！容易使用！在讨论面向对象时，大家总是爱谈可重用性（Reusability），然而过度着重于可重用性（或可复用性）有时会导致过度设计，在考虑面向对象时，易用性（Usability）其实更为重要，在这里将相关的操作与状态放在一起，就是易用性的一个例子。

5.2.3 定义内部属性

当前的 Account 类拥有 name、number 与 balance 三个属性可供存取，虽然设计了 deposit()、withdraw() 方法，用户想要更改 Account 对象的状态时，都要通过这些方法，然而可能会有人如下误用：

```
import bank
acct = bank.Account('Justin', '123-4567', 1000)
acct.balance = 1000000
```

这样的结果显然就没有经过 deposit() 或 withdraw() 的相关条件检查，并且直接修改了 balance 属性的值，如果 IDE 有智能提示功能，如图 5-1 所示可直接看到 name、number、balance 属性，那么用户就可能会直接执行这类违规的存取。

如果想要避免用户直接的误用，那么可以使用 self.__xxx 的方式定义内部值域。例如：

object-oriented3 bank.py

```
class Account:
    def __init__(self, name, number, balance):
        self.__name = name
        self.__number = number
        self.__balance = balance

    def deposit(self, amount):
        if amount <= 0:
            print('存款金额不得为负')
        else:
            self.__balance += amount

    def withdraw(self, amount):
        if amount > self.__balance:
            print('余额不足')
        else:
            self.__balance -= amount

    def __str__(self):
        return "Account('{name}', '{number}', {balance})".format(
            name = self.__name, number = self.__number, balance = self.__balance
        )
```

在 Account 类定义时，可以使用 self.__name、self.__number、self.__balance 来存取属性。然而若用户创建 Account 实例并赋值给 acct，就不能使用 acct.__name、acct.__number、acct.__balance 来进行属性的存取，这会引发 **AttributeError** 错误。若使用的 IDE 有智能提示功能，也不会带出这些属性，如图 5-2 所示。

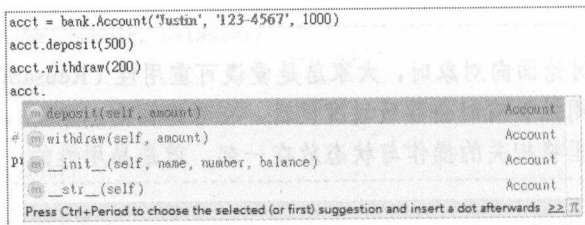


图 5-2 IDE 智能提示不会带出__xxx 的属性

不过, 属性名称前加上_, 只是一种避免直接误用的方式, 当一个属性名称前加上_, 用户仍然可以用另一种方式来存取。例如:

```
acct = bank.Account('Justin', '123-4567', 1000)
print(acct._Account_name)
acct._Account_balance = 1
```

也就是说, 属性若使用__xxx 这样的名称, 则会自动转换为“_类名称__xxx”, Python 并没有完全阻止存取属性, 只要在原来的属性名称前加上_类名称, 依然可以存取到名称为__开头的属性。然而, 我们并不建议这么做, 一个__xxx 名称的属性惯例上是作为类定义时内部相关流程操作之用, 外界最好不要知道其存在, 更别说是操作了。如果真的想这么做, 最好清楚地知道自己在做些什么。

5.2.4 定义外部属性

之前使用了__xxx 这样的格式来定义内部属性, 不过留下了一个问题, 若只想获取账户名称、账号、余额等信息, 以便在相关用户界面上显示, 那该怎么办呢? 不建议以 acct._Account_name 这样的方式, 那还能用什么方式?

基本上, 可以直接定义一些方法来返回 self.__name、self.__number 这些内部属性的值, 例如:

```
class Account:
    ...

    def name(self):
        return self.__name

    def number(self):
        return self.__number

    def balance(self):
        return self.__balance
```

这么一来, 用户就可以使用 acct.name()、acct.number()这样的方式来获取值。对于这种情况, 可以考虑在这类方法上加注@property, 例如:



object-oriented4 bank.py

```
class Account:
    def __init__(self, name, number, balance):
        self.__name = name
        self.__number = number
        self.__balance = balance

    @property
```

```
def name(self):
    return self.__name

@property
def number(self):
    return self.__number

@property
def balance(self):
    return self.__balance
```

其他程序代码同前一范例，故略

这么一来，用户就可以使用 `acct.name`、`acct.number`、`acct.balance` 这样的形式获取值。例如：

object-oriented4 bank_demo.py

```
import bank

acct = bank.Account('Justin', '123-4567', 1000)

acct.deposit(500)
acct.withdraw(200)

print('账户名称: ', acct.name)
print('账户号码: ', acct.number)
print('账户余额: ', acct.balance)
```

然而，目前的程序代码编写无法直接使用 `acct.balance = 10000` 这样的形式来设置属性值，因为 `@property` 只允许 `acct.balance` 这样的形式取值。

提示 >>>

反过来说，如果在程序设计的一开始没有使用 `self.__balance` 的方式，而是以 `self.balance` 定义内部属性，用户也使用 `acct.balance` 来获取值，后来考虑避免进一步被误用，想改用 `self.__balance` 来定义内部属性，这时就可以像上面的范例，定义一个方法并加注 `@property`。如此一来，用户原来的程序代码也不会受到影响，这就是统一存取原则（Uniform access principle¹）的实现。

如果这个范例想要进一步提供 `acct.balance = 10000` 这样的形式，可以使用 `@name.setter`、`@number.setter`、`@balance.setter` 来标注对应的方法。例如：

```
class Account:
    ...
    @name.setter
    def name(self, name):
        # 可实现一些设置值时的条件控制
        self.__name = name

    @number.setter
    def number(self, number):
```

¹ Uniform access principle: en.wikipedia.org/wiki/Uniform_access_principle

```
# 可实现一些设置值时的条件控制
self.__number = number

@balance.setter
def balance(self, balance):
    # 可实现一些设置值时的条件控制
    self.__balance = balance
...
```

被@property 标注的 xxx 取值方法 (Getter) 可以使用 @xxx.setter 标注对应的设值方法 (Setter), 使用 @xxx.deleter 来标注对应的删除值的方法。取值方法返回的值可以是实时运算的结果, 在设置值的方法中必要时可以使用流程语句等来实现一些存取控制。

提示 >>>

设值方法与取值方法在设计时有一些相关的考虑, 如果感兴趣, 可以参考“Getter、Setter 的用与不用”:

openhome.cc/Gossip/Programmer/GetterSetter.html

5.3 类语法的细节

思考面向对象有许多不同的切入角度, 前一节以“粘合”状态与操作为起点, 进一步探讨定义内部属性与外部属性的需求, 并使用了一些简单的 Python 语句来实现, 接下来这一节, 要讨论 Python 中定义类时的更多语句和语法细节。

提示 >>>

如果想从另一个切入角度来了解定义类的需求, 可以参考“何谓封装”:

www.slideshare.net/JustinSDK/java-se-7-16580919

5.3.1 绑定与未绑定方法

在 5.2.2 小节中曾经说过, 定义在类中的方法 (method) 本质上也是函数, 以目前我们定义的 Account 类为例, 可以如下创建实例, 并调用已定义的方法。

```
acct = Account('Justin', '123-4567', 1000)
acct.deposit(500)
acct.withdraw(200)
```

如果试着将 acct.deposit 或 acct.withdraw 赋值给一个变量, 我们会发现, 变量实际上引用着一个函数, 而且可以对函数进行调用。例如:

```
>>> import bank
>>> acct = bank.Account('Justin', '123-4567', 1000)
>>> deposit = acct.deposit
>>> withdraw = acct.withdraw
>>> deposit
<bound method Account.deposit of <bank.Account object at 0x014FB8F0>
>>> withdraw
<bound method Account.withdraw of <bank.Account object at 0x014FB8F0>
>>> deposit(500)
```



```
>>> withdraw(200)
>>> print(acct)
Account(Justin, 123-4567, 1300)
>>>
```

提示 >>>

在上面的例子中,如果直接在 REPL 中键入 `acct`,会显示 `<bank.Account object at 0x0141B8F0>` 这样的字样,这是因为 REPL 中会使用 `repr()` 来获取对象的描述字符串,而我们的 `Account` 类并没有定义 `__repr__()` 方法,因此显示的样式来自于默认的 `__repr__()`。

在这里可以看到,试着显示 `acct.deposit` 或 `acct.withdraw` 的字符串描述时,会出现 'bound method' 这样的字样,这表示此函数是个绑定方法。也就是说,此函数已经绑定了一个 `Account` 实例,也就是 `Account` 类在定义方法时的第一个参数 `self` 引用的实例。

当使用 `acct.deposit(500)` 的方式来调用方法时, `acct` 引用的对象实际上就会传给 `deposit()` 方法的第一个参数。相对地,如果在类中定义了一个方法,没有任何参数会怎样呢?

```
>>> class Some:
...     def nothing():
...         print('nothing')
...
>>> s = Some()
>>> s.nothing()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: nothing() takes 0 positional arguments but 1 was given
>>>
```

如果通过类的实例调用方法时,点运算符左边的对象会传给方法作为第一个参数,然而这里的 `nothing()` 方法不接受任何参数,因此发生了 `TypeError` 错误,并且说明错误在于程序试图在调用时给予一个参数。

相对于绑定方法,像这样定义在类中且没有定义任何参数的方法,称之为未绑定方法 (Unbound method),这类方法充其量只是将类名称作为一种命名空间,可以通过类名称来调用它,或用函数对象进行调用。

```
>>> Some.nothing()
nothing
>>> nothing = Some.nothing
>>> nothing()
nothing
>>>
```

一个有趣的问题是,有没有办法获取绑定方法绑定的对象呢?虽然不鼓励这么做,不过确实可以通过绑定方法的特定属性 `__self__` 来获取绑定的对象。例如:

```
>>> class Some:
...     def me(self):
...         return self
...
>>> s = Some()
>>> s.me() is s.me.__self__
True
>>>
```

提示>>>

从 Python 2.6 开始，绑定方法的 `__self__` 属性就存在了，同时间也可以使用 `im_self` 属性，不过 `im_self` 在 Python 3 之后被剔除了。

5.3.2 静态方法与类方法

如果使用 5.2.4 小节设计的 `bank.py` 中的 `Account` 类来创建一个实例并赋值给 `acct`，当调用 `acct.deposit(500)` 时，会将 `acct` 引用的实例传给 `deposit()` 的第一个 `self` 参数。实际上也可以如下获得相同的效果：

```
acct = Account('Justin', '123-4567', 1000)
Account.deposit(acct, 500)
```

如果想要有类似 `deposit = acct.deposit` 的效果，也可以如下编写：

```
deposit = lambda amount: Account.deposit(acct, amount)
```

🔍 标注 `@staticmethod`

假设我们要在 `Account` 类中增加一个 `default()` 函数以便创建默认账户，只需要指定名称与账号，开户时余额默认为 100 即可。

```
class Account:
    ...
    def default(name, number):
        return Account(name, number, 100)
```

提示>>>

当然，这个需求也可以在 `__init__()` 上使用默认自变量来实现，这里只是为了示范，因而请暂时忘记有默认自变量的存在。

我们原本是希望 `default()` 函数以 `Account` 类作为命名空间，因为它与创建账户有关，而用户应该以 `Account.default('Monica', '765-4321')` 的方式来调用它。然而，若用户如下误用，正好也能够执行。

```
acct = Account('Justin', '123-4567', 1000)
# 显示 Account(Account('Justin', '123-4567', 1000), 1000, 100)
print(acct.default(1000))
```

就这个例子来说，`acct` 引用的对象传给了 `default()` 方法的第一个参数 `name`，而执行的过程正好也没有引发错误，只不过显示了怪异的结果。

如果在定义类时，希望某个方法不被拿来作为绑定方法，可以使用 `@staticmethod` 加以标注。例如：

```
class Account:
    ...
    @staticmethod
    def default(name, number):
        return Account(name, number, 100)
```

这么一来，除了可以使用 `Account.default('Monica', '765-4321')` 的方式来调用它，就算用户通过类的实例来调用它，像 `acct.default('Monica', '765-4321')`，`acct` 也不会被传入作为 `default()` 的第一个参数。

虽然可以通过实例来调用 `@staticmethod` 标注，但是建议通过类名称来调用，明确地让类名称作为静态方法的命名空间。

标注 `@classmethod`

来仔细看看上面的例子，`default()` 方法中编写时固定了 `Account` 这个名称，如果要修改类名称，就要记得修改 `default()` 中的类名称，我们可以让 `default()` 的实现更有弹性。

首先要知道的是，在 Python 中定义的类也会产生对应的对象，这个对象是 `type` 的实例。例如：

```
>>> class Some:
...     pass
...
>>> Some
<class '__main__.Some'>
>>> type(Some)
<class 'type'>
>>> s = Some()
>>> s.__class__
<class '__main__.Some'>
>>> s.__class__()
<__main__.Some object at 0x0143C730>
>>>
```

通过对象的 `__class__` 属性可以得知，该对象是从哪个类构建而来的，也可以通过获取的 `type` 实例来构建对象。

因此，只要能在先前的 `default()` 方法中获取当前所在类的 `type` 实例，就可以不用编写固定的类名称了。对于这个需求，可以在 `default()` 方法上标注 `@classmethod`。例如：

```
class Account:
...
    @classmethod
    def default(cls, name, number):
        return cls(name, number, 100)
```

类中的方法如果标注了 `@classmethod`，那么第一个参数一定是接受所在类的 `type` 实例，因此在 `default()` 方法中可以使用第一个参数来构建对象。同样地，建议通过 `Account.default()` 让 `Account` 成为 `default()` 方法的命名空间。

5.3.3 属性命名空间

到目前为止，读者已经看过几种可以作为命名空间的方式了，这时会想到模块，在上一节我们知道类也可以作为命名空间使用，除此之外呢？每个可作为命名空间的实例都是一个对象，在 5.1.2 小节就介绍过，每个模块导入后都会是一个对象，是 `module` 类的实例，每个类也是一个对象，是 `type` 类的实例。

如果必要，一个自定义的类实例也可以作为命名空间。例如：

```
>>> class Namespace:
```



```
...     pass
...
>>> ns = Namespace()
>>> ns.some = 'Just a value'
>>> ns.other = 'Just another value'
>>>
```

在上面的例子中，`ns` 引用的对象不就是作为 `some` 与 `other` 的命名空间吗？某种程度上，类的实例确实是作为属性的命名空间。

提示

在一些程序设计语言中本身没有提供命名空间的机制，像 JavaScript，开发者为了管理名称，就有不少使用对象来实现类似的机制。

每个对象本身都会有个 `__dict__` 属性，当中记录着类或实例所拥有的特性。例如：

```
>>> class Some:
...     def __init__(self, x):
...         self.x = x
...     def add(self, y):
...         return self.x + y
...
>>> s = Some(10)
>>> s.__dict__
{'x': 10}
>>> Some.__dict__
mappingproxy({'__dict__': <attribute '__dict__' of 'Some' objects>, '__weakref__': <attribute '__weakref__' of 'Some' objects>, '__init__': <function Some.__init__ at 0x01421588>, '__doc__': None, 'add': <function Some.add at 0x01421348>, '__module__': '__main__'})
>>>
```

从中可以看到，真正属于实例的属性其实只有 `x`，`Some` 中定义的 `add` 方法其实属于类。这也用来解释，当调用 `s.add(10)` 时，为何效果相当于 `Some.add(s,10)`，实际上就是通过 `Some` 类调用了 `add` 方法。

在 Python 中，名字中具有两个下划线的方法不建议直接调用，如果想获取 `__dict__` 的数据，那么可以使用 `vars()` 函数。例如：

```
>>> class Ball:
...     PI = 3.14159
...
>>> vars(Ball)
mappingproxy({'__dict__': <attribute '__dict__' of 'Ball' objects>, '__weakref__': <attribute '__weakref__' of 'Ball' objects>, 'PI': 3.14159, '__doc__': None, '__module__': '__main__'})
>>> ball = Ball()
>>> vars(ball)
{}
>>> Ball.PI
3.14159
>>> ball.PI
3.14159
>>>
```

在这里的 `Ball` 类中，直接定义了一个 `PI` 变量，从 `vars(Ball)` 的结果可以看到，这样的变量属于 `Ball` 类，而不是 `ball` 引用的实例，这类变量以类作为命名空间，因此建议通过类名称来存取。然而，确实也能通过 `ball.PI` 这样的方式来取用，当一个实例上找不到对应的属性时，会寻找实例的类，

看看其中有没有对应的属性，如果有就可以取用，如果没有就会发生 `AttributeError` 错误。

提示>>>

更细节的流程其实是，若尝试通过实例获取属性，而实例的 `__dict__` 中没有，则会到产生实例的类的 `__dict__` 中寻找。若在类的 `__dict__` 仍没有找到，则会试着调用 `__getattr__()` 来获取属性。若没有定义 `__getattr__()` 方法，就会发生 `AttributeError` 错误。第 14 章会谈到如何定义 `__getattr__()`。

为什么一再强调，若函数或变量以类为命名空间，建议通过类名称来调用或存取呢？一是语义上比较清楚，一眼就可以看出函数或变量是以类为命名空间；二是还可以避免以下的问题：

```
>>> ball.PI = 3.14
>>> ball.PI
3.14
>>> Ball.PI
3.14159
>>>
```

这里的操作接续了上一个 REPL 的示范，虽然一个实例上找不到对应的属性时，会寻找实例的类，看看其中有没有对应的属性，如果有就可以取用。然而，如果在这样的实例上指定属性值时，就会直接在实例上创建属性，而不是修改实例的类中对应的属性。

既然自定义的类型可以在构建出来的实例上直接新增属性，那么可不可以在类中直接新增方法呢？答案是可以的。

```
>>> class Account:
...     pass
...
>>> acct = Account()
>>> acct.name = 'Justin'
>>> acct.number = '123-4567'
>>> acct.balance = 1000
>>> def deposit(self, amount):
...     self.balance += amount
...
>>> Account.deposit = deposit
>>> acct.deposit(500)
>>> acct.balance
1500
>>>
```

可以看到，就算新增的方法是在实例构建之后，通过实例调用方法时仍是可以生效的。

在 5.1.2 小节曾经谈过 `del`，它可以用来删除变量，或者删除已导入当前模块的名称（本质上也是个变量），也可以用来删除某个对象上的属性。例如：

```
>>> class Some:
...     def __init__(self, x):
...         self.x = x
...     def add(self, y):
...         return self.x + y
...
>>> s = Some(10)
>>> s.x
10
>>> del s.x
```

```
>>> s.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Some' object has no attribute 'x'
>>> del Some.add
>>> Some.add
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Some' has no attribute 'add'
>>>
```

由于模块也是个对象，因此也可以使用 `del` 来删除模块上定义的名称。例如：

```
>>> import math
>>> math.pi
3.141592653589793
>>> del math.pi
>>> math.pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'math' has no attribute 'pi'
>>>
```

其实 `del` 真正的作用是删除某对象上的指定属性。举例来说，当以全局作用域创建变量时，实际上就是在当时的模块对象上创建属性，而在全局作用域使用 `del` 删除变量时，就是从当时的模块对象上删除属性。

提示

每个模块都有个 `__name__` 属性，一个模块被 `import` 时，`__name__` 属性会被设置为模块名称，直接使用 Python 指令执行某模块时，`__name__` 属性会被设置为 `'__main__'`。无论如何，想要获取当前的模块对象，可以使用 `sys.modules[__name__]` 来获取。

5.3.4 定义运算符

到目前为止，我们知道了 `__init__()`、`__str__()`、`__repr__()` 这类方法在 Python 中用来定义某些特定行为。例如，`__init__()` 可定义对象创建后的初始化流程，`__str__()`、`__repr__()` 用于定义对象的字符串描述，之后我们还会看到更多这类的方法定义。

就现在而言，我们可以知道的是，在 Python 中可以使用特定的 `__xxx__()` 方法名称来定义特定类型遇到运算符时应该具有的操作。实际上，第 3 章谈到类型与运算符时，它们彼此间的运算操作就是由这些特定方法来定义的。例如：

```
>>> x = 10
>>> y = 3
>>> x + y
13
>>> x.__add__(y)
13
>>> x % 3
1
>>> x.__mod__(3)
1
>>>
```

可以看到，`+` 运算符实际上是由 `int` 的 `__add__()` 方法定义的，`%` 运算符是由 `int` 的 `__mod__()` 方

法定义的。为了实际了解这些方法是如何定义的，先来看一个具体的范例，创建一个有理数类，并定义其+、-、*、/等运算符的操作。

classes xmath.py

```
class Rational:
    def __init__(self, number, denom)::  ← ❶ 设置分子与分母
        self.number = number
        self.denom = denom

    def __add__(self, that):  ← ❷ 定义 + 运算符
        return Rational(
            self.number * that.denom + that.number * self.denom,
            self.denom * that.denom
        )

    def __sub__(self, that):  ← ❸ 定义 - 运算符
        return Rational(
            self.number * that.denom - that.number * self.denom,
            self.denom * that.denom
        )

    def __mul__(self, that):  ← ❹ 定义 * 运算符
        return Rational(
            self.number * that.number,
            self.denom * that.denom
        )

    def __truediv__(self, that):  ← ❺ 定义 / 运算符
        return Rational(
            self.number * that.denom,
            self.denom * that.denom
        )

    def __str__(self):  ← ❻ 定义 __str__()
        return '{number}/{denom}'.format(
            number = self.number, denom = self.denom
        )

    def __repr__(self):  ← ❼ 定义 __repr__()
        return 'Rational({number}, {denom})'.format(
            number = self.number, denom = self.denom
        )
```

在创建 Rational 实例之后，会使用__init__()初始化分子与分母❶；对象常见的+、-、*、/等操作分别是由__add__()❷、__sub__()❸、__mul__()❹、__truediv__()❺定义（//由__floordiv__()定义的）。至于对象的字符串描述，若要以 1/2 这样的形式显示运算结果，则定义在__str__()方法之中❻，__repr__()方法的实现则采用'Rational(1, 2)'这类字符串描述❼。

来看看 REPL 中的执行结果：

```
>>> import xmath
>>> r1 = xmath.Rational(1, 2)
>>> r2 = xmath.Rational(2, 3)
>>> print(r1 + r2)
7/6
>>> print(r1 - r2)
-1/6
```

```
>>> print(r1 * r2)
2/6
>>> print(r1 / r2)
3/6
>>>
```

这应该能让读者回想起在 3.2.2 小节中曾经谈过的 `decimal.Decimal` 类，该类创建的实例可以直接使用 `+`、`-`、`*`、`/` 进行运算，就是因为 `decimal.Decimal` 类定义了相对应的方法。

类似地，如果想定义 `>`、`≥`、`<`、`≤`、`==`、`!=` 等比较运算，可以分别实现 `__gt__()`、`__ge__()`、`__lt__()`、`__le__()`、`__eq__()` 或 `__comp__()` 等方法。不过，对象的相等性要考虑的要素比较多一些，这里暂不讨论，等到第 6 章再来说明。

5.3.5 `__new__()`、`__init__()` 与 `__del__()`

到目前为止只要谈到 `__init__()`，都是说这个方法是在类的实例构建之后进行初始化的方法，而不是说 `__init__()` 是用来构建类实例的。这样的说法是有意义的，因为类的实例如何构建，实际上是由 `__new__()` 方法来定义，`__new__()` 方法的第一个参数是类本身，之后可定义任意参数作为构建对象之用。

`__new__()` 方法可以返回对象，如果返回的对象是第一个参数的类实例，接下来就会执行 `__init__()` 方法，`__init__()` 方法的第一个参数就是 `__new__()` 返回的对象。`__new__()` 如果没有返回第一个参数的类实例（返回别的实例或 `None`），就不会执行 `__init__()` 方法。

一个简单测试构建与初始化流程的例子如下所示。

```
>>> class Some:
...     def __new__(cls, isClzInstance):
...         print('__new__')
...         if isClzInstance:
...             return object.__new__(cls)
...         else:
...             return None
...     def __init__(self, isClzInstance):
...         print('__init__')
...         print(isClzInstance)
...
>>> Some(True)
__new__
__init__
True
>>> Some(False)
__new__
>>>
```

在上面的示范中可以看到，调用类创建实例时指定的自变量会成为 `__new__()` 与 `__init__()` 的第二个参数，如果有更多自变量就会按序指定给后续参数。

当使用 `Some(True)` 创建实例时，`isClzInstance` 是 `True`，因而执行了 `if` 区块，这时使用 `object.__new__(cls)` 来创建类的实例，而不是直接以 `cls(isClzInstance)` 来创建，这是因为前者只是单纯创建对象，后者等同于再执行了一次 `Some(True)`，这样又会再调用 `__new__()`，然后再执行 `cls(isClzInstance)`，如此不停地递归下去，直到最后发生 `RecursionError` 错误为止。

如果使用了 `Some(False)` 而使得 `__new__()` 返回 `None`，除了不执行 `__init__()` 方法之外，

Some(False)的结果也会是 None。

`__new__()`如果返回第一个参数的类实例,就会执行`__init__()`方法,借助定义`__new__()`方法就可以决定如何构建对象与初始化对象,一个应用的例子如下。

classes xlogging.py

```
class Logger:
    __loggers = {}  # ① 保存已创建的 Logger 实例
    def __new__(cls, name):
        if name not in cls.__loggers:  # ② 如果字典中不存在对应的 Logger 就创建
            logger = object.__new__(cls)
            cls.__loggers[name] = logger
            return logger
        return cls.__loggers[name]  # ③ 否则返回字典中对应的 Logger 实例
    def __init__(self, name):
        if name not in vars(self):
            self.name = name  # ④ 设置 Logger 的名称
    def log(self, message):  # ⑤ 简单模拟日志的行为
        print('{name}: {message}'.format(name = self.name, message = message))
```

这个 `Logger` 类的设计想法是每个指定名称下的 `Logger` 实例只会有一个,因此使用了一个字典来保存已创建的实例①。如果 `Logger('某名称')`调用时会先执行`__new__()`方法,这时检查字典中是否有指定名称的键存在②,若没有,则表示先前没有创建 `Logger` 实例,此时使用 `object.__new__(cls)`来创建对象,并以 `name` 作为键,创建的对象作为值,且保存在字典中,接着返回创建的对象;如果指定名称已有对应的对象就直接返回③。

由于这个范例的`__new__()`都会返回 `Logger` 实例,在`__init__()`方法中,为了不重复设置 `Logger` 实例的 `name` 属性,使用 `vars(self)`获取 `Logger` 实例上的属性列表,并查看 `name` 是否为 `Logger` 实例的属性之一,如果不是,表示这是新建的 `Logger`,为其设置 `name` 属性④。最后为了方便进行示范,定义了一个简单的 `log()`方法来模拟日志(Logging)的操作⑤。

下面是个简单的测试程序。

classes xlogging_demo.py

```
import xlogging

logger1 = xlogging.Logger('xlogging')
logger1.log('一些日志信息....')

logger2 = xlogging.Logger('xlogging')
logger2.log('另外一些日志信息....')

logger3 = xlogging.Logger('xlog')
logger3.log('再来一些日志信息....')

print(logger1 is logger2)
print(logger1 is logger3)
```

程序中 `logger1` 与 `logger2` 引用的对象都是使用 `xlogging.Logger('xlogging')`来获取的,根据 `Logger` 的定义应该会是相同的 `Logger` 实例,由于 `logger3` 使用的名称不同,因此是不同的 `Logger`

实例。执行结果如下：

```
xlogging: 一些日志信息....
xlogging: 另外一些日志信息....
xlog: 再来一些日志信息....
True
False
```

如果一个对象不再被任何名称引用，就无法在程序流程中继续被使用，那么这个对象就是个垃圾，Python 解释器会在适当时候删除这个对象，以回收相关的资源。如果想在对象被删除时自行定义一些清除相关资源的操作，可以执行 `__del__()` 方法。例如：

```
>>> class Some:
...     def __del__(self):
...         print('__del__')
...
>>> s = Some()
>>> s = None
__del__
>>>
```

在这个例子中，原本被 `s` 引用的对象，由于 `s` 被赋值为 `None`，而不再有任何名称的引用。就这个简单的例子来说，该对象马上就被回收资源了，因此可以看到执行了 `__del__()` 方法。

不过对象被回收的时机并不一定，也就无法预期 `__del__()` 会被执行的时机，因此 `__del__()` 中最好只定义一些不急于执行的资源清除操作，如果有些资源清除操作希望能够掌控执行的时机，那么最好定义其他方法，并在必要时明确进行调用。

提示 >>>

除了这一章介绍的几个 `__xxx__` 方法之外，还有更多其他方法各自定义了特定的操作，这之后按各章主题会有相关的介绍，想要提前知道还有哪些方法，可以先看看“Special method names”：docs.python.org/3/reference/datamodel.html#special-method-names

5.4 重点复习

一个 `.py` 文件就是一个模块，这使得模块成为 Python 中最自然的抽象层。

想要知道一个模块中有哪些名称，可以使用 `dir()` 函数。

当 `import` 某个模块而使得指定的 `.py` 文件被加载时，Python 解释器会为它创建一个 `module` 实例，并创建一个模块名称来引用它。`from import` 会将导入模块中的名称引用的值赋给当前模块中创建的新名称。

如果有些变量并不想被 `from import *` 创建同名变量，可以用下划线作为开头。如果模块中定义了 `__all__` 变量，那么就只有名单中的变量才可以被其他模块 `from import *`。可以使用 `del` 将模块名称或者 `from import` 的名称删除。

想要知道当前已加载的 `module` 名称与实例有哪些，可以通过 `sys.modules` 来查看。

`import`、`import as`、`from import` 可以出现在语句能出现的位置，例如 `if...else` 区块或者函数之中，因此能根据不同的情况进行不同的 `import`。

`sys.path` 列表中列出了寻找模块时的路径，列表内容基本上来自几个来源：执行 Python 解释器

时的文件夹、PYTHONPATH 环境变量、Python 安装的标准链接库等文件夹、PTH 文件列出的文件夹。

可以在一个.pth 文件中列出模块搜索路径, PTH 文件的存放位置在不同操作系统并不相同, 可以通过 site 模块的 getsitepackages() 函数获取正确的位置。

如果想将 PTH 文件放置到其他文件夹中, 可以使用 site.addsitedir() 函数在 PTH 文件中新增文件夹的来源。

若想以对象来思考或组织应用程序的操作, 可以从打算将对象的状态与功能“粘”在一起时开始。

在 Python 中可以使用 class 来创建一个专用类。可以将初始化流程使用 __init__() 方法定义在类之中。

方法名称的前后各有两个连接的下划线, 在 Python 中, 这样的名称意味着在类以外的其他位置不要直接调用, 基本上都会有个函数来调用这类方法。

在调用 __init__() 方法时, 创建的类实例会传入作为方法的第一个参数, 虽然第一个参数的名称可以自定义, 然而在 Python 的惯例中, 第一个参数的名称会命名为 self。

在 Python 中, 对象的方法的第一个参数一定是对象本身。

返回对象描述字符串的方法在 Python 中有个特殊名称 __str__() 专门用来定义这个操作。如果执行 str(acct), 就会调用 acct 的 __str__() 方法获取描述字符串并返回, 如果执行 repr(acct), 就会调用 acct 的 __repr__() 方法获取描述字符串并返回。

__str__() 字符串描述主要是给人类看的易懂格式, 而 __repr__() 是给程序、计算机解析用的特定格式 (像对代表日期的字符串解析, 用来创建一个日期对象), 或者是包含调试用的字符串信息。

如果想要避免用户的误用, 可以使用 self.__xxx 的方式定义内部值域。属性若使用 __xxx 这样的名称, 则会自动转换为 “_类名称_xxx”, Python 并没有完全阻止存取属性, 只要在原来的属性名称前加上 _ 类名称, 仍然可以存取到名称为 _ 开头的属性。

被 @property 标注的 xxx 取值方法 (Getter) 可以使用 @xxx.setter 标注对应的设值方法 (Setter), 使用 @xxx.deleter 来标注对应的删除值的方法, 取值方法返回的值可以是实时运算的结果, 设值方法中, 必要时可以使用流程语句等来实现一些存取控制。

定义在类中的方法本质上也是函数。

定义在类中, 没有定义任何参数的方法被称为未绑定方法 (Unbound method), 这类方法充其量只是将类名称作为命名空间, 可以通过类名称来调用它, 或用函数对象进行调用。

如果在定义类时希望某个方法不被拿来作为绑定方法, 可以使用 @staticmethod 加以标注。

虽然可以通过实例来调用 @staticmethod 标注的方法, 但是建议通过类名称来调用, 明确地让类名称作为静态方法的命名空间。

类中的方法如果标注了 @classmethod, 那么第一个参数一定是接受所在类的 type 实例。

想获取 __dict__ 的数据可以使用 vars() 函数。

del 真正的作用是删除某对象上的指定属性。

对象常见的 +、-、*、/ 等操作分别是由 __add__()、__sub__()、__mul__()、__truediv__() 定义 (// 由 __floordiv__() 定义) 的。若想定义 >、≥、<、≤、==、!= 等比较运算, 可以分别实现 __gt__()、__ge__()、__lt__()、__le__()、__eq__() 或 __comp__() 等方法。

`__init__()`是在类的实例构建之后进行初始化的方法，类的实例如何构建？实际上是由 `__new__()`方法来定义。

如果想在对象被删除时自行定义一些清除相关资源的操作，可以执行 `__del__()`方法。

课后练习

实践题

1. 据说创世纪时有一座汉诺塔由三根钻石棒支撑，神在第一根棒上放置了 64 个从小到大排列的金盘，命令僧侣将所有金盘从第一根棒移至第三根棒，搬运过程遵守大盘在小盘下的原则，若每日仅搬一盘，在盘子全数搬至第三根棒后，此塔将毁损。请编写程序，可输入任意盘数，按以上搬运原则显示搬运过程。

2. 如果有一个二维数组代表迷宫，0 表示道路、2 表示墙壁，如下所示。

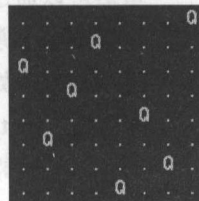
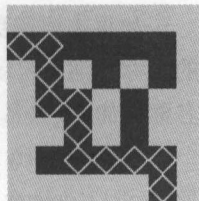
```
maze = [[2, 2, 2, 2, 2, 2, 2],
         [0, 0, 0, 0, 0, 0, 2],
         [2, 0, 2, 0, 2, 0, 2],
         [2, 0, 0, 2, 0, 2, 2],
         [2, 2, 0, 2, 0, 2, 2],
         [2, 0, 0, 0, 0, 0, 2],
         [2, 2, 2, 2, 2, 0, 2]]
```

假设老鼠会从索引(1, 0)开始，请使用程序找出老鼠如何跑至索引(6, 5)位置，并以■代表墙，◇代表老鼠，显示出走迷宫的路径，如右图所示。

3. 有一个 8x8 的棋盘，骑士（即车）走法为国际象棋走法，请编写程序，可指定骑士从棋盘任一位置出发，以标号显示走完所有位置。例如其中一个走法如下。

```
52 21 64 47 50 23 40 3
63 46 51 22 55 2 49 24
20 53 62 59 48 41 4 39
61 58 45 54 1 56 25 30
44 19 60 57 42 29 38 5
13 16 43 34 37 8 31 26
18 35 14 11 28 33 6 9
15 12 17 36 7 10 27 32
```

4. 国际象棋中后可直线和斜线前进，吃掉遇到的棋子。如果棋盘上有八个皇后，请编写程序，显示八个皇后相安无事地放置在棋盘上的所有方式。右图为其中一个放法。



第 6 章

类的继承

学习目标

- 了解继承的目的
- 认识鸭子类型
- 重新定义方法
- 认识 Object 类
- 创建、寻找文件资源



6.1 何谓继承

在面向对象程序设计中，子类继承（Inherit）父类，避免重复定义的行为与实现。不过，并非为了避免重复定义行为与实现就要使用继承，滥用继承而导致程序维护上的问题时有发生，如何正确判断使用继承的时机，以及继承之后如何活用多态，才是学习继承时的重点。

6.1.1 继承共同行为

继承基本上就是避免多个类之间重复定义相同的行为。以实际的例子来说明比较清楚，假设我们正在开发一款 RPG（Role-playing game，角色扮演）游戏，一开始设置的角色有剑士与魔法师。首先定义了剑士类：

```
class SwordsMan:
    def __init__(self, name, level, blood):
        self.name = name # 角色名称
        self.level = level # 角色等级
        self.blood = blood # 角色血量

    def fight(self):
        print('挥剑攻击')

    def __str__(self):
        return "{name}, {level}, {blood}".format(**vars(self))

    def __repr__(self):
        return self.__str__()
```

剑士拥有名称、等级与血量等属性，可以挥剑攻击，为了方便显示剑士的属性，定义了__str__()方法，并让__repr__()的字符串描述直接返回__str__()的结果。

__str__()方法中直接使用了 5.2.3 小节介绍过的 vars()函数，以字典获取当前实例的属性名称与值，然后用 4.2.2 小节介绍过的字典拆解方式，将字典的属性名称与值拆解后，传给字符串的 format()方法，这样的写法和下面的程序相比，后者显然简洁了许多。

```
class SwordsMan:
    ...
    def __str__(self):
        return "{name}, {level}, {blood}".format(
            name = self.name, level = self.level, blood = self.blood)
    ...
```

接着为魔法师定义类：

```
class Magician:
    def __init__(self, name, level, blood):
        self.name = name # 角色名称
        self.level = level # 角色等级
        self.blood = blood # 角色血量

    def fight(self):
        print('魔法攻击')
```

```
def cure(self):
    print('魔法治疗')

def __str__(self):
    return "('{name}', {level}, {blood})".format(**vars(self))

def __repr__(self):
    return self.__str__()
```

注意到什么吗？因为只要是游戏中的角色，都会具有角色名称、等级与血量，也定义了相同的 `__str__()` 与 `__repr__()` 方法，Magician 中粗体字部分与 SwordsMan 中相对应的程序代码重复了。

重复在程序设计上就是不好的信号。举个例子来说，如果要将 name、level、blood 更改为其他名称，那就要修改 SwordsMan 与 Magician 两个类，如果有更多类具有重复的程序代码，那就要修改更多类，造成维护上的不便。

如果要改进，可以把相同的程序代码提升 (Pull up) 至父类 Role，并让 SwordsMan 与 Magician 类都继承自 Role 类，

game1 rpg.py

class Role: ← ① 定义类 Role

```
def __init__(self, name, level, blood):
    self.name = name # 角色名称
    self.level = level # 角色等级
    self.blood = blood # 角色血量

def __str__(self):
    return "('{name}', {level}, {blood})".format(**vars(self))

def __repr__(self):
    return self.__str__()
```

class SwordsMan(Role): ← ② 继承父类 Role

```
def fight(self):
    print('挥剑攻击')
```

class Magician(Role): ← ③ 继承父类 Role

```
def fight(self):
    print('魔法攻击')
```

```
def cure(self):
    print('魔法治疗')
```

在这个范例中定义了 Role 类①，可以看到没什么特别之处，只不过是先前的 SwordsMan 与 Magician 中重复的程序代码都定义在 Role 类中。

接着 SwordsMan 类在定义时，类名称旁边多了个括号，并指定了 Role，这在 Python 中代表着 SwordsMan 继承了 Role②已定义的程序代码，接着 SwordsMan 中定义了自己的 fight() 方法。类似地，Magician 也继承了 Role 类③，并且定义了自己的 fight() 与 cure() 方法。

如何看出确实有继承了呢？以下简单的程序可以看出：

game1 rpg_demo.py

```
import rpg
```



```
swordsman = rpg.SwordsMan('Justin', 1, 200)
print('SwordsMan', swordsman)

magician = rpg.Magician('Monica', 1, 100)
print('Magician', magician)
```

在执行 `print('剑士', swordsman)` 与 `print('魔法师', magician)` 时, 会调用 `swordsman` 与 `magician` 的 `__str__()` 方法, 虽然在 `SwordsMan` 与 `Magician` 类的定义中, 并没有看到定义 `__str__()` 方法, 但是它们都从 `Role` 继承下来了, 因此可以如范例中所示的那样直接使用。执行的结果如下:

```
SwordsMan ('Justin', 1, 200)
Magician ('Monica', 1, 100)
```

可以看到, `__str__()` 返回的字符串描述确实是 `Role` 类中定义的结果。继承的好处之一就是如果要将 `name`、`level`、`blood` 改为其他名称, 那就只需修改 `Role` 类的程序代码就可以了, 只要是继承 `Role` 的子类都无需修改。

6.1.2 鸭子类型

现在有个需求, 请设计一个可以播放角色属性与攻击动画的函数。在 3.2.1 小节讨论变量时曾经说明过, Python 的变量本身没有类型, 若想通过变量操作对象的某个方法, 只要确认该对象上确实有该方法即可, 因此可以如下编写程序:



game2 rpg_demo.py

```
import rpg

def draw_fight(role):
    print(role, end = '')
    role.fight()

swordsman = rpg.SwordsMan('Justin', 1, 200)
draw_fight(swordsman)

magician = rpg.Magician('Monica', 1, 100)
draw_fight(magician)
```

在 `draw_fight()` 函数中直接调用了 `role` 的 `fight()` 方法, 如果是 `fight(swordsman)`, 那么 `role` 就引用 `swordsman` 的实例, 这时 `role.fight()` 相当于 `swordsman.fight()`。同样地, 如果是 `fight(magician)`, `role.fight()` 就相当于 `magician.fight()`。执行结果如下:

```
('Justin', 1, 200) 挥剑攻击
('Monica', 1, 100) 魔法攻击
```

别因为这一章是在讨论继承, 就误以为必须是继承才能有这样的行为, 实际上就这个范例而言, 只要对象上拥有 `fight()` 方法就可以传入 `draw_fight()` 函数。例如:

```
>>> from rpg_demo import draw_fight
(Justin, 1, 200) 挥剑攻击
(Monica, 1, 100) 魔法攻击
>>> class Duck:
...     pass
```

```
...
>>> duck = Duck()
>>> duck.fight = lambda: print('呱呱')
>>> draw_fight(duck)
<__main__.Duck object at 0x0182B410>呱呱
>>>
```

可以看到，在这里就算随便定义一个 Duck 类，创建一个实例，临时指定一个 lambda 函数给 fight 属性，仍然可以传给 draw_fight() 函数执行。由于 Duck 并没有定义 __str__()，因此使用的是默认的 __str__() 实现，因而我们可以看到 <__main__.Duck object at 0x0182B410> 的结果。

在 3.2.1 小节就说过了，这就是动态类型语言界流行的鸭子类型 (Duck typing¹)：“如果它走路像个鸭子，游泳像个鸭子，叫声像个鸭子，那它就是鸭子。”反过来说，虽然这是只鸭子，但是它打起架来像个 Role (具有 fight())，那它就是一个 Role。

鸭子类型实际的意义在于：“思考对象的行为，而不是对象的种类。”按照此思维设计的程序，会具有比较高的通用性。就像在这里看到的 draw_fight() 函数，不仅仅能接受 Role 类与子类实例，只要是具有 fight() 方法的实例，draw_fight() 都能接受。

提示 >>>

确实某些特定的情况下，还是免不了要判断对象的种类，并给予不同的流程。不过在多数的情况下，应优先选择思考对象的行为。

6.1.3 重新定义方法

上一节的 draw_fight() 函数若传入 SwordsMan 或 Magician 实例时，各自会显示 ('Justin', 1, 200) 挥剑攻击或 ('Monica', 1, 100) 魔法攻击，如果想要显示 SwordsMan('Justin', 1, 200) 挥剑攻击或 Magician('Monica', 1, 100) 魔法攻击，要怎么做呢？

读者也许会想要判断传入的对象到底是 SwordsMan 还是 Magician 的实例，然后分别显示剑士或魔法师的字样，在 Python 中，确实有个 isinstance() 函数可以进行这类的判断。例如：

```
def draw_fight(role):
    if isinstance(role, rpg.SwordsMan):
        print('SwordsMan', end = '')
    elif isinstance(role, rpg.Magician):
        print('Magician', end = '')

    print(role, end = '')
    role.fight()
```

不过，这并不是个好主意，若是未来有更多角色，则势必要增加更多类型检查的判断式，在多数情况下，检查类型给予不同的流程行为对于程序的维护有着不良的影响，应该避免。

那么该怎么做呢？print(role, end = "") 时，既然实际上是获取 role 引用实例的 __str__() 返回的字符串并显示，目前 __str__() 的行为是定义在 Role 类中并继承下来，那么可否分别重新定义 SwordsMan 与 Magician 的 __str__() 行为，让它们各自能增加剑士或魔法师的字样呢？

¹ Duck typing: en.wikipedia.org/wiki/Duck_typing

我们可以这么做，不过并不用单纯地在 SwordsMan 或 Magician 中定义以下的 `__str__()`。

```
...
def __str__(self):
    return "SwordsMan('{name}', {level}, {blood})".format(**vars(self))
...
def __str__(self):
    return "Magician('{name}', {level}, {blood})".format(**vars(self))
```

因为 Role 的 `__str__()` 返回的字符串只要各自在前面附上剑士或魔法师就可以了，在继承后如果打算基于父类的方法实现来重新定义某个方法，那么可以使用 `super()` 来调用父类方法。例如：



game2 rpg.py

```
class Role:
    ...
    def __str__(self):
        return '({name}, {level}, {blood})'.format(**vars(self))

    def __repr__(self):
        return self.__str__()

class SwordsMan(Role):
    def fight(self):
        print('挥剑攻击')

    def __str__(self):  ← ① 重新定义类 SwordsMan 的 __str__()
        return 'SwordsMan' + super().__str__()

class Magician(Role):
    def fight(self):
        print('魔法攻击')

    def cure(self):
        print('魔法治疗')

    def __str__(self):  ← ② 重新定义类 Role 的 __str__()
        return 'Magician' + super().__str__()
```

在重新定义 SwordsMan 的 `__str__()` 方法时①，调用了 `super().__str__()`，这会执行父类 Role 中定义的 `__str__()` 方法并返回字符串，这个字符串与'剑士'串接，就是我们想要的结果。同样地，在重新定义 Magician 的 `__str__()` 方法时②，也是使用 `super().__str__()` 获得结果，然后串接'魔法师'字符串。

其实 `super()` 是在类的 `__mro__` 属性中寻找指定的方法，6.2.1 小节与 6.2.5 小节还有针对 `super()` 的探讨。

6.1.4 定义抽象方法

在 6.1.2 小节讨论鸭子类型时曾经谈到，若想通过变量操作对象的某个方法，只要确认该对象上确实有该方法即可，并不一定要在程序代码上有继承的关系。然而有时候，我们希望提醒或强制

子类一定要实现某个方法,或许是怕其他开发者在实现时打错了方法名称(像 `fight()` 打成了 `figth()`),或许是有太多行为必须实现,不小心遗漏了其中一两个。

如果是子类在继承之后一定要实现的方法,可以在父类中指定 `metaclass` 为 `abc` 模块的 `ABCMeta` 类,并在指定的方法上标注 `abc` 模块的 `@abstractmethod` 来达到目的。例如,若想强制 `Role` 的子类一定要实现 `fight()` 方法,可以如下编写:

```
game3 rpg.py
from abc import ABCMeta, abstractmethod  ← ❶ 导入 ABCMeta 与 abstractmethod 名称

class Role(metaclass=ABCMeta):          ← ❷ 指定 metaclass 为 ABCMeta
    def __init__(self, name, level, blood):
        self.name = name # 角色名称
        self.level = level # 角色等级
        self.blood = blood # 角色血量

    @abstractmethod  ← ❸ 标注 @abstractmethod
    def fight(self):
        pass

    def __str__(self):
        return "({name}, {level}, {blood})".format(**vars(self))

    def __repr__(self):
        return self.__str__()

...
```

由于 `ABCMeta` 类与 `abstractmethod` 函数定义在 `abc` 模块中,因此使用 `from import` 将其导入❶,接着在定义 `Role` 类时,指定 `metaclass` 为 `ABCMeta` 类❷, `metaclass` 是个协议,当定义类时指明 `metaclass` 的类时,Python 会在解析完类定义后使用指定的 `metaclass` 来进行类的构建与初始化,这是高级议题,在第 14 章还会说明,目前先当它是个魔法。

接着,我们在 `fight()` 方法上标注了 `@abstractmethod`❸,由于 `Role` 只是个通用的父类,并不知道具体的各个角色会如何进行攻击,也就不需要使用相关的程序代码实现,因此直接在 `fight()` 方法的本体中使用 `pass`。

提示 >>>

在 Python 中, `abc` 或 `ABC` 这个字样,是指 `Abstract Base Class`,也就是抽象基类。通常这些类已实现了一些基础行为,开发者可根据需求使用不同的 `ABC` 来实现想要的功能,但又不用一切从无到有亲手打造。

一旦定义了 `Role` 类,就不能使用 `Role` 来构建对象了,否则会发生 `TypeError` 错误。例如:

```
>>> import rpg
>>> rpg.Role('Justin', 1, 200)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Role with abstract methods fight
>>>
```

如果有一个类继承了 Role 类，没有实现 fight() 方法，在实例化时就会发生 TypeError 错误。

```
>>> Monster('Pika', 3, 500)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Monster with abstract methods fight
>>>
```

然而，由于之前的 SwordsMan 与 Magician 已经实现了 fight() 方法，因此可以顺利地拿来构建对象。

6.2 继承语法的细节

上一节介绍了继承的基础概念与语法，然而结合 Python 的特性，继承还有许多细节必须了解清楚，像必要时怎么调用父类方法、如何定义对象间的 Rich comparison 方法（有人译为“富比较”或“厚比较”）、多重继承的属性查找等，这些将于本节中详细说明。

6.2.1 初识 object 与 super()

在 6.1.1 小节的 rpg.py 中可以看到，SwordsMan 与 Magician 继承了 Role 之后，并没有重新定义自己的 __init__() 方法，因此在构建 SwordsMan 或 Magician 实例时会直接使用 Role 中定义的 __init__() 方法来进行初始化。

还记得在 5.3.5 小节中说明过，类的实例如何构建实际上是由 __new__() 方法来定义吗？那么在没有定义时，__new__() 方法又是由谁提供呢？答案就是 object 类。在 Python 中定义一个类时，如果没有指定父类，那么就是继承 object 类，这个类中提供了一些属性定义，所有的类都会继承这些属性定义。

```
>>> dir(object)
['_class_', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__']
>>>
```

除了 __new__()、__init__() 方法之外，我们曾经接触过的方法有 __str__()、__repr__()，这是用来定义对象的字符串描述的。如果我们定义了一个类，而没有定义 __str__() 或 __repr__()，就会使用 object 默认的字符串描述定义，这个字符串描述会像 6.1.2 小节曾经看过的 <__main__.Duck object at 0x0182B410> 字样一样，意义不大。

其他属性的作用或方法的定义方式之后会在适当的章节进行说明，例如稍后会介绍 __eq__() 与 __hash__() 方法的作用与定义方式，这与对象相等性有关联。

简单来说，在 Python 中如果没有定义的方法，某些场合下必须调用时，就会查看父类中是否有定义的方法。如果定义了自己的方法，那么就会以自己定义的方法为主，而不会主动调用父类的方法。例如：

```
>>> class P:
...     def __init__(self):
```

```

...     print('P __init__')
...
>>> class S(P):
...     def __init__(self):
...         print('S __init__')
...
>>> s = S()
S __init__
>>>

```

在上面的例子中，类 S 继承了 P，并定义了自己的 `__init__()` 方法，在构建 S 的实例时，只调用了 S 中定义的 `__init__()`，而没有调用 P 中的 `__init__()`。有时候这样的行为会是我们所想要的。不过有时候，必须在初始化的过程中也执行父类中定义的初始化。

举个例子来说，如果创建了一个 Account 类，当中定义了 `__name__`、`__number` 与 `__balance` 三个属性，分别代表账户的名称、账号与余额，其中 `__init__()` 方法必须初始化这三个属性。后来又定义了一个 SavingsAccount 类，增加了利率 `__interest_rate` 属性，如果不想在 SavingsAccount 中重复定义初始化 `__name__`、`__number` 与 `__balance` 的过程，那么就会希望直接调用 Account 类中的 `__init__()`。

在 6.1.3 小节中介绍过，可以使用 `super()` 来调用父类中已定义的某个方法，这在 `__init__()` 中当然也是可行的。例如：

inheritance bank.py

```

class Account:
    def __init__(self, name, number, balance):
        self.__name = name
        self.__number = number
        self.__balance = balance
    略...

    def __str__(self):
        return "Account('{name}', '{number}', {balance})".format(
            name = self.__name, number = self.__number, balance = self.__balance
        )

class SavingsAccount(Account):
    def __init__(self, name, number, balance, interest_rate):
        super().__init__(name, number, balance)  ← ① 调用父类的 __init__()
        self.__interest_rate = interest_rate

    def __str__(self):
        return (super().__str__()[0:-1] +
                str(self.__interest_rate) + ')')  ← ② 调用父类的 __str__()

```

由于 SavingsAccount 的 `__init__()` 方法中，使用 `super().__init__(name, number, balance)` 主动调用了 Account 父类的 `__init__()` ①，因此最后构建的对象也会具有 `__name__`、`__number` 与 `__balance` 三个属性。类似地，在 SavingsAccount 的 `__str__()` 中也使用 `super().__str__()` 先获取父类的结果 ②，再加上利率的描述字符串。

如果使用以下的程序进行测试：

inheritance bank_demo.py

```
import bank

savingsAcct = bank.SavingsAccount('Justin', '123-4567', 1000, 0.02)

savingsAcct.deposit(500)
savingsAcct.withdraw(200)

print(savingsAcct)
```

将会有以下的执行结果：

```
Account('Justin', '123-4567', 13000.02)
```

在 Python 3 中，定义方法时使用无自变量的 `super()` 调用等同于 `super(__class__, <first argument>)` 调用，`__class__` 代表当前所在类，`<first argument>` 是指当前所在方法的第一个自变量。

因此就绑定方法来说，在定义方法时使用无自变量的 `super()` 调用，方法的第一个参数名称为 `self` 就相当于 `super(__class__, self)`，将 `bank.py` 中的 `super().__init__(name, number, balance)` 改成 `super(__class__, self).__init__(name, number, balance)`，以及将 `super().__str__()` 改成 `super(__class__, self).__str__()`，执行的结果是相同的。

调用 `super(__class__, <first argument>)` 时，会查找在 `__class__` 的父类中是否有指定的方法，如果有，就将 `<first argument>` 作为调用方法时的第一个自变量。在刚才的 `bank.py` 例子中，`super(__class__, self).__str__()` 会在 `SavingsAccount` 的父类 `Account` 中找到 `__str__()` 方法，结果就相当于以 `Account.__str__(self)` 的方式调用。

实际上，确实也可以在程序代码中直接以 `Account.__init__(self, name, number, balance)`、`Account.__str__(self)` 的方式调用父类中定义的方法，不过这样做缺乏弹性，将来如果修改父类名称，那么子类中的程序代码也就要做出相应的修正，使用 `super().__init__(name, number, balance)`、`super().__str__()` 这样的方式显然更具有弹性，也更加便利。

实际上，`super()` 指定自变量时并不限于在方法之中才能使用，而且是在 `__mro__` 中查找指定的方法，在 6.2.6 小节我们还会看到 `super()` 的进一步探讨。

6.2.2 Rich comparison 方法

在 `object` 类中还定义了 `__lt__()`、`__le__()`、`__eq__()`、`__ne__()`、`__gt__()`、`__ge__()` 等方法，这组方法定义了对对象之间使用 `<`、`≤`、`==`、`!=`、`>`、`≥` 等比较时应该得到的比较结果，这组方法在 Python 官方文件上被称为 Rich comparison 方法（“富比较”或“厚比较”）。

❶ 定义 `__eq__()`

想要使用 `==` 来比较两个对象是否相等，必须定义 `__eq__()` 方法，因为 `__ne__()` 默认会调用 `__eq__()` 并反相其结果（即对结果求反），因此定义了 `__eq__()` 就等于定义了 `__ne__()`，也就可以使用 `!=` 比较两个对象是否不相等。

`object` 定义的 `__eq__()` 方法默认使用 `is` 来比较两个对象，也就是看两个对象实际上是不是同一个实例，所以要比较实质相等性，必须自行重新定义。举个简单的例子，比较两个 `Cat` 对象是否实

际上代表同一只 Cat 的数据。

```
class Cat:
    ...
    def __eq__(self, other):
        # other 引用的就是这个对象，当然是同一对象
        if self is other:
            return True

        # 检查是否有相同的属性，没有的话就不用比了
        if hasattr(other, 'name') and hasattr(other, 'birthday'):
            # 定义如果名称与生日一样，表示两个对象实质上相等
            return self.name == other.name and self.birthday == other.birthday

        return False
```

第二个 if 中使用了 `hasattr()` 函数，它可用来检查指定的对象中是否具有指定的属性。在动态类型语言中，通常会对属性进行检查，较少对类进行检查，这样做可以获得较大的弹性。如果想更严格地检查是否为 Cat 类，那么第二个 if 中可改用 `isinstance(other, __class__)` 来取代。

这里仅示范了 `__eq__()` 实现的基本概念，实际上实现 `__eq__()` 并非这么简单，实现 `__eq__()` 时通常也会实现 `__hash__()`，其原因会等到谈到 Python 标准数据结构链接库时再进行说明，如果读者现在就想知道 `__eq__()` 与 `__hash__()` 实现时要注意的一些事项，可以先参考“对象相等性”。

openhome.cc/Gossip/Python/ObjectEquality.html

► 定义 `__gt__()`、`__ge__()`

想要使用 `>`、`>=`、`<`、`<=` 来进行对象的比较，必须定义 `__gt__()`、`__ge__()`、`__lt__()`、`__le__()` 方法。然而由于 `__lt__()` 与 `__gt__()` 互补，`__le__()` 与 `__ge__()` 互补，因此基本上只要定义 `__gt__()`、`__ge__()` 就可以了。来看个简单的例子：

```
>>> class Some:
...     def __init__(self, value):
...         self.value = value
...     def __gt__(self, other):
...         return self.value > other.value
...     def __ge__(self, other):
...         return self.value >= other.value
...
>>> s1 = Some(10)
>>> s2 = Some(20)
>>> s1 > s2
False
>>> s1 >= s2
False
>>> s1 < s2
True
>>> s1 <= s2
True
>>> 2
```

► 使用 `functools.total_ordering`

并不是每个对象都要定义整组比较方法。如果真的需要定义整组方法的行为，那么可以使用 `functools.total_ordering`。例如：

```
>>> from functools import total_ordering
>>> @total_ordering
... class Some:
...     def __init__(self, x):
...         self.x = x
...     def __eq__(self, other):
...         return self.x == other.x
...     def __gt__(self, other):
...         return self.x > other.x
...
>>> s1 = Some(10)
>>> s2 = Some(20)
>>> s1 >= s2
False
>>> s1 <= s2
True
>>>
```

当一个类被标注了`@total_ordering` 时, 必须实现`__eq__()`方法, 并选择`__lt__()`、`__le__()`、`__gt__()`、`__ge__()`其中一个方法来实现, 这样就可以拥有整组的比较方法了, 其背后的基本原理是只要定义了`__eq__()`以及`__lt__()`、`__le__()`、`__gt__()`、`__ge__()`其中一个方法, 假设是`__gt__()`, 那么剩下的`__ne__()`、`__lt__()`、`__le__()`、`__ge__()`就可以各自调用这两个方法来完成比较的行为, 稍后在 6.2.4 小节会看到一个类似的实现。

6.2.3 使用 enum 枚举

在 Python 中如果想要枚举值, 可以通过字典或者类来定义。例如使用字典的情况:

```
>>> Action = {
...     'stop': 1,
...     'right': 2,
...     'left': 3,
...     'up': 4,
...     'down': 5
... }
>>> Action['stop']
1
>>> Action['down']
5
>>>
```

或者使用类定义的方式:

```
>>> class Action:
...     stop = 1
...     right = 2
...     left = 3
...     up = 4
...     down = 5
...
>>> Action.right
2
>>> Action.left
3
>>>
```

基本上这两种方式是可以解决问题的。不过问题在于无法检查枚举值是否重复, 可以通过 `Action['up'] = 5` 或者是 `Action.up = 5` 这样的方式来修改枚举值。如果通过类方式来定义, `Action`

类本身还能够实例化，这些都是使用上的困扰。

从 Python 3.4 开始新增了 `enum` 模块，其中提供了 `Enum`、`IntEnum` 等类，可以用来继承以便定义枚举类。继承 `Enum`，枚举值可以是各种类型，不过建议使用状态不可变的值（例如字符串），继承 `IntEnum`，枚举值就只能是整数。例如：

```
>>> from enum import IntEnum
>>> class Action(IntEnum):
...     stop = 1
...     right = 2
...     left = 3
...     up = 4
...     down = 5
...
>>> Action.left
<Action.left: 3>
>>> Action()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() missing 1 required positional argument: 'value'
>>> Action.left = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Program Files (x86)\Python35-32\lib\enum.py", line 305, in __setattr__
    raise AttributeError('Cannot reassign members.')
AttributeError: Cannot reassign members.
>>>
```

可以看到，我们无法使用 `Action()` 来创建一个对象，也无法重新指定枚举值。实际上，`Action()` 用来指定枚举值，然后返回枚举对象，枚举对象中具有 `name` 与 `value`，可用来获取枚举名称与枚举值，也可以使用 `[]` 指定枚举名称来获取枚举对象。例如：

```
>>> Action(3)
<Action.left: 3>
>>> enum_member = Action(3)
>>> enum_member.name
'left'
>>> enum_member.value
3
>>> Action['left']
<Action.left: 3>
>>>
```

继承了 `Enum` 或 `IntEnum` 而定义的类可以使用 `for in` 来迭代枚举。

```
>>> for member in Action:
...     print(member.name, '\t:', member.value)
...
stop      : 1
right     : 2
left      : 3
up        : 4
down      : 5
>>>
```

继承 `Enum` 或 `IntEnum` 类定义枚举时，枚举名称不得重复，然而枚举值可以重复。例如：

```
>>> class Action(IntEnum):
...     stop = 1
...     stop = 2
...
>>>
```

```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in Action
  File "C:\Program Files (x86)\Python35-32\lib\enum.py", line 66, in __setitem__
    raise TypeError('Attempted to reuse key: %r' % key)
TypeError: Attempted to reuse key: 'stop'
>>> class Action(IntEnum):
...     stop = 1
...     left = 1
...
>>> Action(1)
<Action.stop: 1>
>>> Action['left']
<Action.stop: 1>
>>>

```

如果枚举名称不同而值相同，那么后者会是前者的别名，因此就上例来说，无论使用 `Action(0)` 或 `Action['left']`，一律返回 `<Action.stop: 1>`。

如果想要在枚举时值不得重复，可以在类中加注 `enum` 模块的 `@unique`，这么一来若枚举时有重复的值，就会引发 `ValueError` 错误。例如：

```

>>> from enum import IntEnum, unique
>>> @unique
... class Action(IntEnum):
...     stop = 1
...     right = 2
...     left = 3
...     up = 4
...     down = 4
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "C:\Program Files (x86)\Python35-32\lib\enum.py", line 567, in unique
    (enumeration, alias_details))
ValueError: duplicate values found in <enum 'Action'>: down -> up
>>>

```

`enum` 模块¹的官方帮助文档中还有一些关于枚举的相关说明，如果有兴趣可以进一步参考那些帮助文档。

6.2.4 多重继承

在 `Python` 中可以进行多重继承，也就是一次继承两个父类的程序代码定义，父类之间使用逗号分隔开。多个父类继承下来的方法名称没有冲突时是最简单的情况，例如：

```

>>> class P1:
...     def mth1(self):
...         print('mth1')
...
>>> class P2:
...     def mth2(self):
...         print('mth2')
...

```

¹ `enum` 模块：docs.python.org/3/library/enum.html

```
>>> class S(P1, P2):
...     pass
...
>>> s = S()
>>> s.mth1()
mth1
>>> s.mth2()
mth2
>>>
```

然而，如果继承时多个父类中有相同的方法名称，就要注意搜索的顺序，基本上是从子类开始寻找名称，接着是同一层级父类从左到右搜索，再到更上层同一层级父类从左到右搜索，直到到达顶层为止。例如：

```
>>> class P1:
...     def mth(self):
...         print('P1 mth')
...
>>> class P2:
...     def mth(self):
...         print('P2 mth')
...
>>> class S1(P1, P2):
...     pass
...
>>> class S2(P2, P1):
...     pass
...
>>> s1 = S1()
>>> s2 = S2()
>>> s1.mth()
P1 mth
>>> s2.mth()
P2 mth
>>>
```

在上面的例子中，由于 S1 继承父类的顺序是 P1、P2，S2 是 P2、P1，因此在寻找 mth() 方法时 S1 实例使用的是 P1 继承而来方法，S2 使用的是 P2 继承而来的方法。

具体来说，一个子类在寻找指定的属性或方法名称时会根据类的 `__mro__` 属性的元组 (tuple) 中元素的顺序进行寻找 (MRO 全名是 Method Resolution Order，即方法解析顺序)，若想要知道直接父类，则可以通过类的 `__bases__` 来得知父类是哪个或者哪些。

```
>>> S1.__mro__
(<class '__main__.S1'>, <class '__main__.P1'>, <class '__main__.P2'>, <class 'object'>)
>>> S1.__bases__
(<class '__main__.P1'>, <class '__main__.P2'>)
>>> S2.__mro__
(<class '__main__.S2'>, <class '__main__.P2'>, <class '__main__.P1'>, <class 'object'>)
>>> S2.__bases__
(<class '__main__.P2'>, <class '__main__.P1'>)
>>>
```

`__mro__` 是只读属性，有趣的是可以通过改变 `__bases__` 来改变直接父类，从而使得 `__mro__` 的内容也跟着变动。例如：

```
>>> S2.__bases__ = (P1, P2)
>>> S2.__mro__
(<class '__main__.S2'>, <class '__main__.P1'>, <class '__main__.P2'>, <class 'object'>)
>>> s2.mth()
```



```
P1 mth
>>>
```

在上面的例子中，故意调换了 S2 的父类为 P1、P2 的顺序，结果 `__mro__` 查找父类的顺序也变成了 P1 在前、P2 在后，因此这次通过 S2 实例调用 `mth()` 方法时先找到的是 P1 上的 `mth()` 方法。

如果定义类时 python 解释器无法生成 `__mro__`，会引发 `TypeError` 错误，一个简单的例子如下：

```
>>> class First:
...     pass
...
>>> class Second(First):
...     pass
...
>>> class Third(First, Second):
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Cannot create a consistent method resolution
order (MRO) for bases Second, First
>>>
```

在 6.1.4 小节谈过如何定义抽象方法，如果有一个父类中定义了抽象方法，而另一个父类中实现了一个方法，且名称与前一个父类的抽象方法相同，那么子类继承这两个父类的顺序会决定抽象方法是否得到实现。例如：

```
>>> from abc import ABCMeta, abstractmethod
>>> class P1(metaclass=ABCMeta):
...     @abstractmethod
...     def mth(self):
...         pass
...
>>> class P2:
...     def mth(self):
...         print('mth')
...
>>> class S(P1, P2):
...     pass
...
>>> s = S()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class S with abstract methods mth
>>> class S(P2, P1):
...     pass
...
>>> s = S()
>>> s.mth()
mth
>>>
```

基本上，判定一个抽象方法是否有实现也是按照 `__mro__` 中类的顺序，如果在 `__mro__` 中先找到有类实现了方法，后续才找到定义了抽象方法的类，那么就会认定已经实现了抽象方法。

6.2.5 创建 ABC（抽象基类）

在 Python 中可以多重继承，这是个双刃剑，特别是在继承的父类中，具有相同名称的方法定

义时尽管有 `__mro__` 属性可以作为名称搜索的依据，然而总是会令情况变得复杂。

多重继承的能力通常建议只用来继承 ABC，也就是抽象基类（Abstract Base Class）。一个抽象基类不会定义属性，也不会有 `__init__()` 定义。

在什么样的情况下，需要定义一个符合刚才要求的抽象基类呢？来考虑一个 `Ball` 类，其中定义了一些比较大小的方法（暂时忘了 6.2.2 小节介绍过的 `functools.total_ordering`）。

```
class Ball:
    def __init__(self, radius):
        self.radius = radius
    def __eq__(self, other):
        return hasattr(other, 'radius') and self.radius == other.radius
    def __gt__(self, other):
        return hasattr(other, 'radius') and self.radius > other.radius
    def __ge__(self, other):
        return self > other or self == other
    def __lt__(self, other):
        return not (self > other and self == other)
    def __le__(self, other):
        return (not self >= other) or self == other
    def __ne__(self, other):
        return not self == other
```

提示 >>>

虽然在 6.2.2 小节介绍过，`__eq__()` 与 `__ne__()` 互补，`__gt__()` 与 `__lt__()` 互补，`__ge__()` 与 `__le__()` 互补，只需实现 `__eq__()`、`__gt__()`、`__ge__()` 即可，这里为了突显可复用的共同实现，也将互补的方法实现出来了。

事实上，许多对象都会用到比较方法，仔细观察以上的程序代码，我们会发现一些可复用的方法，可以将之抽离出来。

inheritance xabc.py

```
from abc import ABCMeta, abstractmethod
```

```
class Ordering(metaclass=ABCMeta):
```

```
    @abstractmethod
```

```
    def __eq__(self, other):
```

```
        pass
```

```
    @abstractmethod
```

```
    def __gt__(self, other):
```

```
        pass
```

```
    def __ge__(self, other):
```

```
        return self > other or self == other
```

```
    def __lt__(self, other):
```

```
        return not (self > other and self == other)
```

```
    def __le__(self, other):
```

```
        return (not self >= other) or self == other
```

```
    def __ne__(self, other):
```

```
        return not self == other
```

① 定义抽象方法

② 定义可复用的共同实现

像 Ordering 这样的类就是一个抽象基类，由于实际的对象 `==` 以及 `>` 的行为，必须根据不同的对象而有不同的实现，在 Ordering 中不予以定义，必须由子类继承之后加以实现，在这里使用 `@abstractmethod` 标注不是必要的^❶。然而为了避免开发者在继承之后忘了实现必要的方法，使用 `@abstractmethod` 标注具有提醒的作用。至于 `__ge__()`、`__lt__()`、`__le__()`、`__ne__()` 方法只是从刚才的 Ball 类中抽取出来的可复用实现^❷。

提示 >>>

可以看到，Python 的 abc 模块提供了 ABCMeta、abstractmethod 等用来定义抽象基类的组件。

有了这个 Ordering 类之后，若有对象需要比较的行为，只要继承 Ordering 并实现 `__eq__()` 与 `__gt__()` 方法就可以了。例如，刚才的 Ball 类现在只需如下编写：



inheritance xabc.py

```
from xabc import Ordering

class Ball(Ordering):  ← ❶ 继承 Ordering
    def __init__(self, radius):
        self.radius = radius

    def __eq__(self, other):  ← ❷ 实现 __eq__() 与 __gt__()
        return hasattr(other, 'radius') and self.radius == other.radius

    def __gt__(self, other):
        return hasattr(other, 'radius') and self.radius > other.radius

b1 = Ball(10)
b2 = Ball(20)

print(b1 > b2)
print(b1 <= b2)
print(b1 == b2)
```

在继承了 Ordering 之后^❶，Ball 类只需要实现 `__eq__()` 与 `__gt__()` 方法^❷，就能具有比较的行为。

由于 Python 可以多重继承，在必要时可以同时继承多个 ABC，针对必要的方法进行实现，同时可以拥有多个 ABC 类中早已定义好的其他可复用实现。实际上，在 Python 的标准链接库中就提供了不少 ABC，之后的章节会看到其中一些 ABC 的介绍。

提示 >>>

像这种抽离可复用程序片段，必要时能以某种方式安插至类定义内的特性中，有时被称为 Mix-in（混入）。

6.2.6 探讨 super()

在多数情况下，只需要在定义方法时使用无自变量的 `super()` 来调用父类中的方法就足够了。然而，在 6.2.1 小节中也谈到无自变量的 `super()` 调用，其实是 `super(__class__, <first argument>)` 的简便方法，这表示 `super()` 其实是可以使用具有自变量的调用方式。接下来我们就要探讨这些具有自

变量的 `super()`调用是怎么一回事，这是高级主题，若暂时不感兴趣，读者可以先跳过这部分，待日后有机会再回过头来看看。

在 6.2.1 小节中提过，在一个绑定方法中使用无自变量 `super()`调用时，如果绑定方法的第一个参数名称是 `self`，就相当于使用 `super(__class__, self)`。如果在 `@classmethod` 标注的方法中无自变量调用 `super()`呢？在 5.3.2 小节中说明过，`@classmethod` 标注的方法第一个参数一定是绑定类本身，如果参数名称是 `clz`，那么无自变量调用 `super()`就相当于调用 `super(__class__, clz)`。

因此，我们也可以在 `@classmethod` 标注的方法中直接使用无自变量 `super()`调用，这相当于调用父类中以 `@classmethod` 标注定义的方法。

```
>>> class P:
...     @classmethod
...     def cmth(clz):
...         print('P', clz)
...
>>> class S(P):
...     @classmethod
...     def cmth(clz):
...         super().cmth()
...         print('S', clz)
...
>>> S.cmth()
P <class '__main__.S'>
S <class '__main__.S'>
>>>
```

如果使用 `help(super)`查看帮助文档，就会看到 `super()`的几种调用方式。

```
>>> help(super)
Help on class super in module builtins:

class super(object)
 | super() -> same as super(__class__, <first argument>)
 | super(type) -> unbound super object
 | super(type, obj) -> bound super object; requires isinstance(obj, type)
 | super(type, type2) -> bound super object; requires issubclass(type2, type)
 | Typical use to call a cooperative superclass method:
 | ...
```

无自变量的调用方式已经说明过多次了，直接来看看 `super(type, obj)`，可以看到这个调用方式必须符合 `isinstance(obj, type)`，也就是 `obj` 必须是 `type` 的实例。在一个绑定方法中使用 `super()`时相当于 `super(__class__, self)`，`self` 是当时类 `__class__` 的一个实例。

另一个 `super(type, type2)`的调用方式必须符合 `issubclass(type2, type)`，也就是 `type2` 必须是 `type` 的子类，有趣的是 `issubclass(type, type)`的结果也是 `True`。在 `@classmethod` 标注的方法中，使用 `super()`调用时相当于 `super(__class__, clz)`。

在 6.2.1 小节中曾谈到，调用 `super(__class__, <first argument>)`时，会查找在 `__class__` 的父类中是否有指定的方法，如果有，就将 `<first argument>`作为调用方法时的第一个自变量。更具体地说，

调用 `super(type, obj)` 时会使用 `obj` 的类的 `__mro__` 列表, 从指定 `type` 的下一个类开始查找, 看看是否有指定的方法, 如果有, 就将 `obj` 当作调用方法的第一个自变量。

因此, 在一个多层的继承体系或者具有多重继承的情况下, 通过 `super(type, obj)` 我们可以指定要调用哪个父类中的绑定方法。例如:

```
>>> class P:
...     def mth(self):
...         print('P')
...
>>> class S1(P):
...     def mth(self):
...         print('S1')
...
>>> class S2(P):
...     def mth(self):
...         print('S2')
...
>>> class SS(S1, S2):
...     pass
...
>>> ss = SS()
>>> super(SS, ss).mth()
S1
>>> super(S1, ss).mth()
S2
>>> super(S2, ss).mth()
P
>>>
```

在上面的例子中, `P`、`S1`、`S2` 都定义了 `mth()` 方法, `SS` 继承了 `S1` 与 `S2`, `ss` 的类是 `SS`, 其 `__mro__` 中类的顺序为 `SS`、`S1`、`S2`、`P`。

调用 `super(SS, ss).mth()` 时会从 `SS` 下一个类开始寻找 `mth()` 方法, 结果就是使用 `S1` 的 `mth()` 方法; 调用 `super(S1, ss).mth()` 时会从 `S1` 的下一个类开始寻找 `mth()` 方法, 结果就是使用 `S2` 的 `mth()` 方法; `super(S2, ss).mth()` 时会从 `S2` 的下一个类开始寻找 `mth()` 方法, 结果就是使用 `P` 的 `mth()` 方法。调用 `super(type, type2)` 时会使用 `type2` 的 `__mro__` 列表, 从指定 `type` 的下一个类开始查找, 看看是否有指定的方法, 如果有, 就将 `type2` 当作调用方法的第一个自变量。

因此, 可以模仿刚才的范例, 针对 `@classmethod` 标注的方法做个类似的测试过程。

```
>>> class P:
...     @classmethod
...     def cmth(cls):
...         print('P', cls)
...
>>> class S1(P):
...     @classmethod
...     def cmth(cls):
...         print('S1', cls)
...
>>> class S2(P):
...     @classmethod
...     def cmth(cls):
...         print('S2', cls)
...
>>> class SS(S1, S2):
...     pass
...
>>>
```

```
>>> super(SS, SS).cmth()
S1 <class '__main__.SS'>
>>> super(S1, SS).cmth()
S2 <class '__main__.SS'>
>>> super(S2, SS).cmth()
P <class '__main__.SS'>
>>>
```

由于这里使用了@classmethod，读者可能会问，@staticmethod 标注的方法是什么呢？如果想要调用父类的静态方法，其实也是要使用 super(type, type2)的形式，例如：

```
>>> class P:
...     @staticmethod
...     def smth(p):
...         print(p)
...
>>> class S(P):
...     pass
...
>>> super(S, S).smth(10)
10
>>>
```

由于@staticmethod 标注的方法是一个未绑定方法，在 help(super)的帮助说明中我们可以看到 super(type)→unbound super object 的字样，这会让人以为可以使用 super(S).smth(10)，然而 super() 返回的是个代理对象，是 super 的实例而不是类本身，因此 super(S).smth(10)会发生 AttributeError 错误。

使用 super(type)的机会非常少，一种可能性是作为描述器（Descriptor）使用，因为 super(type) 返回的对象会具有 __get__()、__set__()方法，因此会有以下的执行结果。

```
>>> class P:
...     def mth(self):
...         print('P')
...
>>> class S(P):
...     pass
...
>>> s = S()
>>> super(S).__get__(s, S).mth()
P
>>>
```

提示 >>>

本书第 14 章会介绍描述器，届时读者会知道描述器的定义方式，也就能明白 __get__() 的意义，基本上可以忽略 super(type) 的用法，若想要深入了解，可以参考“Things to Know About Python Super”中的说明：

www.artima.com/weblogs/viewpost.jsp?thread=236275

www.artima.com/weblogs/viewpost.jsp?thread=236278

www.artima.com/weblogs/viewpost.jsp?thread=237121

6.3 文档与软件包资源

如果读者跟随本书学习到这一节，也就对函数、模块与类的定义与编写都有了一定的认识。

一个好的程序设计语言必须有好的链接库来搭配，而一个好的链接库必定要有清楚、详尽的文档，对 Python 来说正是如此。在这一节中，将先从如何编写文档开始，之后来看看如何查询现有的文档，以及去哪里寻找 Python 社区贡献的软件包。

6.3.1 DocStrings

对于链接库的使用，实际上 Python 的标准链接库源码本身就附有文档。以 `list()` 来说，如果在 REPL 中键入 `list.__doc__` 会发生什么事呢？

```
>>> list.__doc__
"list() -> new empty list\nlist(iterable) -> new list initialized from iterable's items"
>>>
```

这字符串很奇怪，还有一些换行字符？如果键入 `help(list)` 就不会觉得奇怪了。

```
>>> help(list)
Help on class list in module builtins:

class list(object)
| list() -> new empty list
| list(iterable) -> new list initialized from iterable's items
|
略...
```

实际上，`help()` 函数会获取 `list.__doc__` 的字符串，结合一些它在链接库中查找出来的信息，然后加以显示。通过 `__doc__` 获得的字符串称为 DocStrings，可以自行在源码中定义。例如想为自定义函数定义 DocStrings，可以如下：

```
def max(a, b):
    '''Given two numbers, return the largest one.'''
    return a if a > b else b
```

在函数、类或模块定义的一开头，使用 `'''` 括起来的多行字符串，会成为函数、类或模块的 `__doc__` 属性值，也就会成为 `help()` 的输出内容之一。先来看个函数的例子：

```
>>> def max(a, b):
...     '''Given two numbers, return the largest one.'''
...     return a if a > b else b
...
>>> max.__doc__
'Given two numbers, return the largest one.'
>>> help(max)
Help on function max in module __main__:

max(a, b)
    Given two numbers, return the largest one.

>>>
```

如果 DocStrings 只有一行，那么 `'''` 括起来的字符串就不会换行。如果 `'''` 包括换行与缩排，那么 `__doc__` 的内容也会包括这些换行与缩排。

因此，按照惯例单行的 DocStrings 会在一行中使用 `'''` 左右括起来，而函数或方法中的 DocStrings 若是多行字符串，`'''` 紧接的第一行会是个函数的简短描述，之后空一行后才是参数或其他相关说明，最后换一行并缩排作为结束，这样在 `help()` 输出时会比较美观。例如：

```
>>> def max(a, b):
...     '''Find the maximum number.
...
...     Given two numbers, return the largest one.
...     '''
...     return a if a > b else b
...
>>> help(max)
Help on function max in module __main__:

max(a, b)
    Find the maximum number.

    Given two numbers, return the largest one.

>>>
```

如果是类或模块的多行 DocStrings, 会在"'"的后面马上换行, 然后以相同层次缩排, 并开始编写说明。例如:

docs openhome/abc.py

```
# Copyright 2016 openhome.cc. All Rights Reserved.
# Permission to use, copy, modify, and distribute this code and its
# documentation for educational purpose.
```

```
"""
```

```
Abstract Base Classes (ABCs) for Python Tutorial
```

```
Just a demo for DocStrings.
```

```
"""
```

```
from abc import ABCMeta, abstractmethod
```

```
class Ordering(metaclass=ABCMeta):
```

```
    '''
```

```
    A abc for implementing rich comparison methods.
```

```
    The class must define __gt__() and __eq__() methods.
```

```
    '''
```

```
    @abstractmethod
```

```
    def __eq__(self, other):
```

```
        '''Return a == b'''
```

```
        pass
```

```
    @abstractmethod
```

```
    def __gt__(self, other):
```

```
        '''Return a > b'''
```

```
        pass
```

```
    def __ge__(self, other):
```

```
        '''Return a >= b'''
```

```
        return self > other or self == other
```

```
...
```

如果想为软件包编写 DocStrings, 可以在软件包所对应的文件夹下的 `__init__.py` 文件内使用"'"括起要编写的字符串说明。例如:

```
docs openhome/__init__.py
```

```
'''
Libraries of openhome.cc

If you can't explain it simply, you don't understand it well enough.
    - Albert Einstein
'''
```

在 REPL 中，可针对软件包使用 `help()`。例如：

```
>>> import openhome.abc
>>> help(openhome)
Help on package openhome:

NAME
    openhome - Libraries of openhome.cc

DESCRIPTION
    If you can't explain it simply, you don't understand it well enough.
        - Albert Einstein

PACKAGE CONTENTS
    abc

FILE
    c:\workspace\docs\openhome\__init__.py

>>>
```

至于刚才定义的 `abc.py` 模块，使用 `help()` 的结果如下：

```
>>> help(openhome.abc)
Help on module openhome.abc in openhome:

NAME
    openhome.abc - Abstract Base Classes (ABCs) for Python Tutorial

DESCRIPTION
    Just a demo for DocStrings.

CLASSES
    builtins.object
        Ordering

    class Ordering(builtins.object)
        | A abc for implementing rich comparison methods.
        | The class must define __gt__() and __eq__() methods.
        | Methods defined here:
        |     __eq__(self, other)
        |         Return a == b
        |     __ge__(self, other)
        |         Return a >= b
    略...
```

也可以直接使用 `help(openhome.abc.Ordering)`、`help(openhome.abc.Ordering.__eq__)` 来分别查询

所对应的 DocStrings。如果想要进一步认识 DocStrings 的编写或使用惯例,可以参考标准链接库(位于 Python 安装目录的 Lib 文件夹中)源码中的编写方式,或者是参考 PEP 275:

www.python.org/dev/peps/pep-0257/#what-is-a-docstring

6.3.2 查询官方文档

除了使用 `help()` 查询文档中对于 Python 官方的链接库,也可以通过网络在线来查询,文档地址是 `docs.python.org`,默认会显示最新版本的 Python 文档,可以在首页左上角选择想要的版本,如图 6-1 所示。

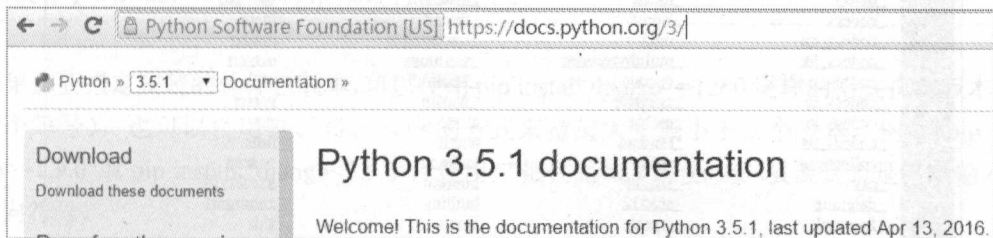


图 6-1 Python 官方文档

提示 >>>

在 1.2.3 小节中谈过,Windows 版本的 Python 安装文件夹内提供了一个 `python351.chm` 文件,其中包含了许多 Python 文档,便于开发者随时查阅。

在官方文档中,有关于链接库 API 查询的部分列在 Indices and tables 中,在编写 Python 程序的过程中经常需要在这里查询相关的 API 如何使用,如图 6-2 所示。

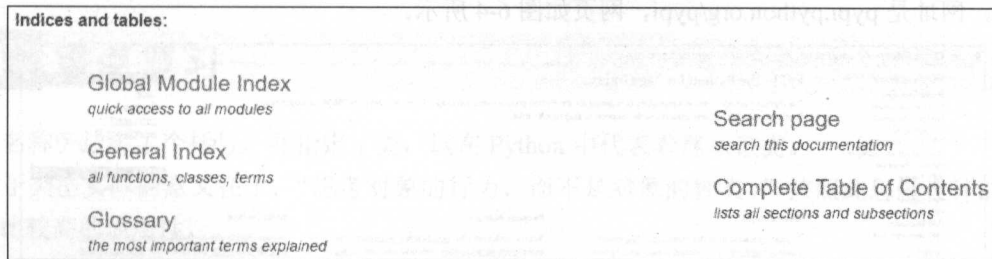


图 6-2 API 索引供查询之用

除了连到网站查询官方 API 文档之外,还可以使用内建的 `pydoc` 模块启动一个简单的 `pydoc` 服务器,例如:

```
>python -m pydoc -p 8080
Server ready at http://localhost:8080/
Server commands: [b]rowser, [q]uit
server>
```

在执行 `python` 时指定 `-m` 自变量,表示执行指定模块中顶层的程序流程,在这里是指定执行 `pydoc` 模块,并附上 `-p` 自变量指定了 8080,这会创建一个简单的文档服务器,并在 8080 端口接受

连接，如图 6-3 所示。我们可以选择 b 启动默认的浏览器，或自行启动浏览器来连接 <http://localhost:8080/>，之后就可以进行文档查询了。

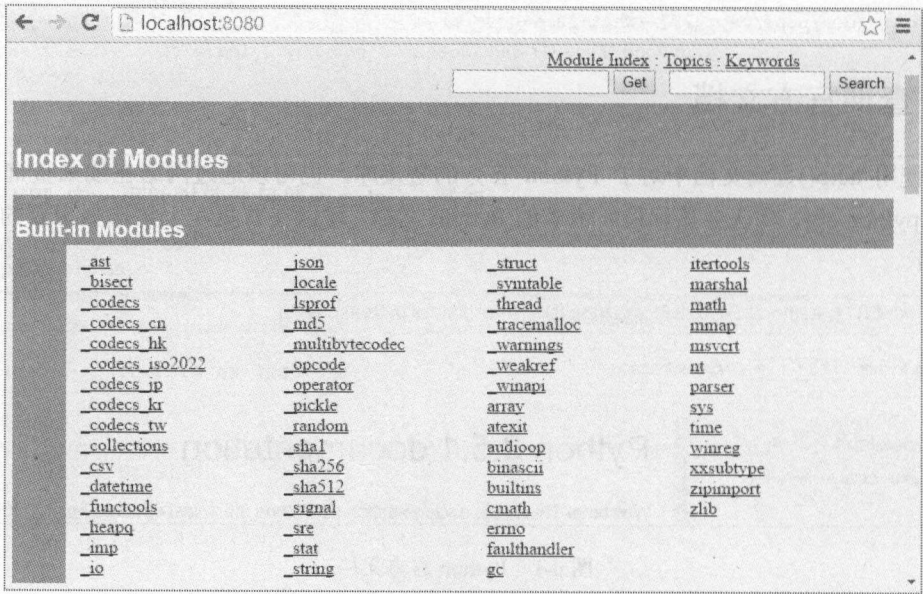


图 6-3 pydoc 文档服务器

6.3.3 PyPI 与 pip

如果标准链接库无法满足你的需求，Python 社区中还有数量庞大的链接库在等着你。如果想要寻找第三程序库，那么 Python 官方维护的 **PyPI (Python Package Index)** 网站可以作为不错的起点，网址是 pypi.python.org/pypi，网页如图 6-4 所示。

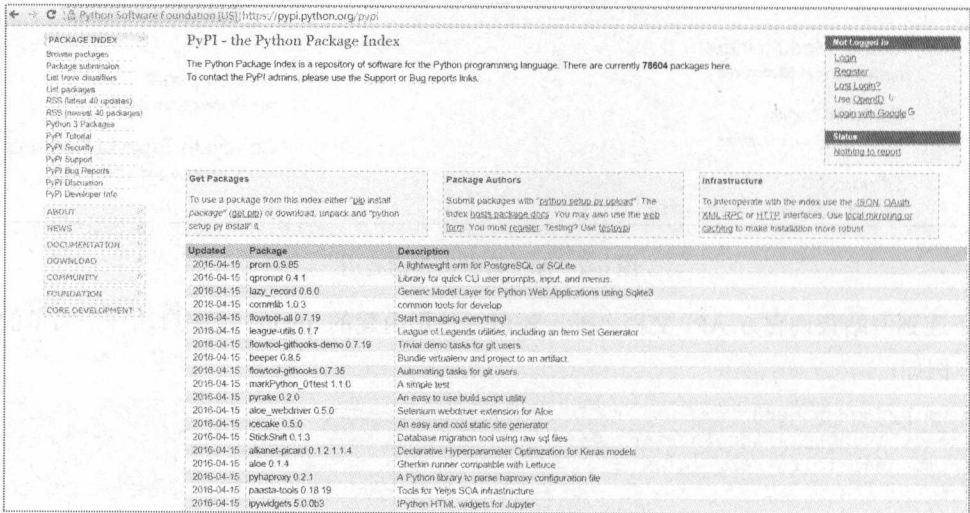


图 6-4 PyPI 网站

读者可以在 PyPI 上寻找符合自己需求的软件包，想要安装上面的软件包，可以通过 **pip** 来安

装。从 Python 3.4 开始就内建了 `pip` 指令（也可以使用 `python -m pip` 来执行安装）。想要使用 `pip` 安装指定的软件包，可以使用 `pip install '软件包名称'`，想要删除则使用 `pip uninstall '软件包名称'`。例如想要安装 Web 开发框架，可以使用 `pip install django`，即安装最新版本的 Django，若想删除，则可以使用 `pip uninstall django`。

提示 >>>

如果在 Windows 中将 Python 安装在 C:\Program Files (x86) 中，需要系统管理员的身份，只有管理员的权限才能使用 `pip` 安装或卸载软件包。

由于 `pip` 本身在不断更新，首次执行 `pip` 会检查版本，若 `pip` 指令不是最新的就出现提示，可以执行 `pip install --upgrade pip` 来进行更新。

如果想要指定安装的软件包版本，可以使用 `pip install django==1.9.0` 这样的格式指定版本号（注意是两个等号），也可以使用 `>=`、`<=`、`>`、`<` 的方式来指定大于或小于某个版本，例如 `pip install "django>=1.9.0"` 或 `pip install "django<1.9.0"`，使用 `"` 括住是为了避免 `>` 或 `<` 被误认为是标准输入输出的重定向符号。

如果想要一次安装多个软件包，可以在一个文本文件中编写软件包需求，例如在一个 `requirements.txt` 中编写：

```
django=1.9.0
flask>0.10.1
numpy
```

接着只要执行 `pip -r install requirements.txt` 就可以按文件中列出的软件包进行安装。如果想要知道更多 `pip` 的使用细节，可以参考 `pip` 的帮助文档：

```
pip.pypa.io/en/stable/
```

6.4 重点复习

类名称旁边多了个括号，并指定了类，这在 Python 中代表着继承该类。

鸭子类型实际的意义在于：“思考对象的行为，而不是对象的种类。”按照此思维设计的程序会具有比较高的通用性。

在继承后若打算基于父类的方法实现重新定义某个方法，可以使用 `super()` 来调用父类方法。

如果希望子类在继承之后一定要实现的方法，可以在父类中指定 `metaclass` 为 `abc` 模块的 `ABCMeta` 类，并在指定的方法上标注 `abc` 模块的 `@abstractmethod`。抽象类不能用来实例化，继承了抽象类而没有实现抽象方法的类，也不能用来实例化。

如果没有指定父类，那么就是继承 `object` 类。

在 Python 中如果没有定义的方法，某些场合下又必须调用时，就会看看父类中是否有定义，如果定义了自己的方法，那么就会以自己定义的方法为主，而不会主动调用父类的方法。

在 Python 3 中，在定义方法时使用无自变量的 `super()` 调用等同于 `super(__class__, <first argument>)` 调用，`__class__` 代表着当前所在类，而 `<first argument>` 是指当前所在方法的第一个自变量。

就绑定方法来说，在定义方法时使用无自变量的 `super()` 调用，方法的第一个参数名称为 `self`，就相当于 `super(__class__, self)`。

在 `object` 类中定义了 `__lt__()`、`__le__()`、`__eq__()`、`__ne__()`、`__gt__()`、`__ge__()` 等方法，这组方法定义了对象之间使用 `<`、`<=`、`==`、`!=`、`>`、`>=` 等比较时应该得到的比较结果。这组方法在 Python 官方文件上被称为 Rich comparison 方法（“富比较”或“厚比较”）。

想要使用 `==` 来比较两个对象是否相等，必须定义 `__eq__()` 方法，因为 `__ne__()` 默认会调用 `__eq__()` 并反相其结果（即对结果求反），因此定义了 `__eq__()` 就等于定义了 `__ne__()`，也就可以使用 `!=` 比较两个对象是否不相等。

`object` 定义的 `__eq__()` 方法默认使用 `is` 来比较两个对象，实现 `__eq__()` 时通常也会实现 `__hash__()`。

由于 `__lt__()` 与 `__gt__()` 互补，`__le__()` 与 `__ge__()` 互补，因此基本上只要定义 `__gt__()`、`__ge__()` 就可以了。

从 Python 3.4 开始新增了 `enum` 模块。

在 Python 中可以进行多重继承，也就是一次继承两个父类的程序代码定义，父类之间使用逗号分隔开。

一个子类在寻找指定的属性或方法名称时，会根据类的 `__mro__` 属性的元组（元组）中元素的顺序进行寻找（MRO 全名是 Method Resolution Order，即方法解析顺序）。若想要知道直接父类，则可以通过类的 `__bases__` 来得知父类是哪个或者哪些。

判定一个抽象方法是否有实现，也是按照 `__mro__` 中类的顺序。

多重继承的能力通常建议只用来继承 ABC，也就是抽象基类（Abstract Base Class），一个抽象基类不会定义属性，也不会有 `__init__()` 定义。

在函数、类或模块定义的一开头，使用 `"""` 括起来的多行字符串会成为函数、类或模块的 `__doc__` 属性值，也就会成为 `help()` 的输出内容之一。

在执行 `python` 时指定 `-m` 自变量表示执行指定模块中顶层的程序流程。

Python 官方维护的 **PyPI**（Python Package Index）网站可以作为搜索链接库时不错的起点。

从 Python 3.4 开始就内建了 `pip` 指令。

课后练习

实践题

1. 虽然目前不知道要采用的执行环境是文本模式、图形界面或者是 Web 页面，现在要编写出一个猜数字游戏，随机产生 0 到 9 的数字，程序可获取用户输入的数字，并与随机产生的数字相比，如果相同就显示“猜中了”，如果不同就继续让用户输入数字，直到猜中为止。请问你该怎么做？

2. 在 5.3.4 小节曾经开发过一个 `Rational` 类，请为该类加上 Rich comparison 方法的实现，让它可以有 `>`、`>=`、`<`、`<=`、`==`、`!=` 的比较能力。

第 7 章

例外处理

学习目标

- 使用 try、except 处理例外
- 认识例外继承结构
- 认识 raise 的使用时机
- 运用 finally 清除资源
- 使用 with as 管理资源

```
workspace\pdb_demo>python -m pdb filter_demo2.py
(Pdb)>
```

```
> def filter_it(predicate, lt):
    """Filter a list of elements based on a predicate function.
    Args:
        predicate: A function that takes an element and returns a boolean.
        lt: A list of elements to be filtered.
    Returns:
        A new list containing only the elements that passed the predicate.
    """
    result = []
    for elem in lt:
        if predicate(elem):
            result.append(elem)
    return result
```

```
1. def filter_it(predicate, lt):
```

```
2.     result = []
```

```
3.     for elem in lt:
```

```
4.         if predicate(elem):
```

```
5.             result.append(elem)
```

```
6.     return result
```

```
8. def len_greater_than(num):
```

```
9.     """Return a list of elements whose length is greater than num.
    Args:
        num: A number representing the minimum length.
    Returns:
        A list of elements whose length is greater than num.
    """
```

```
10.     return len_greater_than(num)
```

```
11.     return len_greater_than(num)
```

```
(Pdb)>
```



7.1 语法与继承结构

当某些原因使得执行流程无法继续时，在 Python 中可以引发（Raise）例外（Exception），至今为止看过的 `TypeError`、`AttributeError`、`ValueError` 等错误就是具体的例子。Python 可以在例外发生时加以处理，可以自行引发例外，或者是自定义例外。

7.1.1 使用 try、except

来看一个简单的程序，用户可以连续输入整数，最后输入 0 结束后会显示输入数的平均值。

exceptions average.py

```
numbers = input('输入数字 (用空格隔开): ').split(' ')
print('平均值', sum(int(number) for number in numbers) / len(numbers))
```

如果用户正确地输入每个整数，程序会如预期地显示平均值。

```
输入数字 (用空格隔开): 10 20 30 40
平均值 25.0
```

如果用户不小心输入时出现了错误，那就会出现奇怪的信息，例如第三个数输入为 3o，而不是 30 了。

```
输入数字 (用空格隔开): 10 20 3o 40
Traceback (most recent call last):
  File "C:/workspace/exceptions/average.py", line 2, in <module>
    print('平均值', sum((int(number) for number in numbers)) / len(numbers))
  File "C:/workspace/exceptions/average.py", line 2, in <genexpr>
    print('平均值', sum((int(number) for number in numbers)) / len(numbers))
ValueError: invalid literal for int() with base 10: '3o'
```

这段错误信息对调试是很有价值的，不过先看到错误信息的最后一行。

ValueError: invalid literal for int() with base 10: '3o'

问题的根源在于 `int()` 接收了一个 '3o' 的字符串，无法将这样的字符串解析为一个整数，在不指定基数的情况下，`int()` 预期的字符串必须是十进制整数。

如果只是想要处理调用 `int()` 时的 `ValueError` 错误，可以编写一条 `if...else` 语句来检查传入的每个字符串，看它们是否是以 10 为底的整数，例如定义一个 `all_int_str()` 之类的函数。

```
numbers = input('输入数字 (用空格隔开): ').split(' ')
if all_int_str(numbers):
    print('平均值', sum((int(number) for number in numbers)) / len(numbers))
else:
    print('必须输入整数')
```

这样的方式基本上可行，只不过没有说明是哪个输入发生了错误，既然刚才看到 `ValueError` 的错误信息中提供了发生错误的原因，为何不直接使用上面的信息呢？

exceptions average2.py

```
try:
    numbers = input('输入数字 (用空格隔开): ').split(' ')
    print('平均值', sum(int(number) for number in numbers) / len(numbers))
except ValueError as err:
    print(err)
```

这里使用了 **try**、**except** 语法，python 解释器尝试执行 **try** 区块中的程序代码，如果发生例外，执行流程会跳离例外发生点，然后对比 **except** 声明的类型，看看是否符合引发的例外对象类型，如果是，就执行 **except** 区块中的程序代码。

一个执行无误的范例如下所示：

```
输入数字 (用空格隔开): 10 20 30 40
平均值 25.0
```

范例中如果执行 **int()** 时发生 **ValueError**，流程就会跳到类型声明为 **ValueError** 的 **except** 区块，执行完 **except** 区块后就没有其他程序代码了，所以程序就结束了。一个执行时输入有误的范例如下所示：

```
输入数字 (用空格隔开): 10 20 3o 40
invalid literal for int() with base 10: '3o'
```

当 **int()** 遇到无法解析为整数的字符串时，它无法继续执行程序，因而以例外的方式让调用的客户端得知发生了无法执行程序的错误。然而在 **Python** 中，例外并不一定是错误，例如使用 **for in** 语句时，其实底层就运用到了例外处理机制。

实际上，只要是具有 **__iter__()** 方法的对象，都可以使用 **for in** 来迭代。**__iter__()** 方法应该返回一个迭代器 (Iterator)，该迭代器具有 **__next__()** 方法，每次迭代时就会返回下一个对象，而没有下一个元素时，则会引发 **StopIteration** 例外，通知客户端因为没有下一个可迭代的对象，迭代流程无法继续。

可以使用 **iter()** 方法调用对象的 **__iter__()** 获取迭代器，使用 **next()** 来调用迭代器的 **__next__()** 方法。例如：

```
>>> iterator = iter([10, 20, 30])
>>> next(iterator)
10
>>> next(iterator)
20
>>> next(iterator)
30
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

for in 会在遇到 **StopIteration** 时静静地结束迭代，因此我们不会看到 **StopIteration**。如果使用函数来实现模拟，流程会是：

exceptions for_in.py

```
def for_in(iterable, do_it):
```

```

iterator = iter(iterable)
try:
    while True:
        do_it(next(iterator))
except StopIteration:
    pass

for_in([10, 20, 30], print)

```

在这里看到 `except` 对比到 `StopIteration` 之后并没有使用 `as`，这是因为在发生 `StopIteration` 时不用做什么事，因此不需要使用 `as` 将例外对象指定给某个名称，只要静静地 `pass` 过去就可以了。

`except` 之后可以使用元组 (tuple) 指定多个对象，也可以有多个 `except`，如果没有指定 `except` 后的对象类型，表示捕捉所有引发的对象。举例来说，下面的范例中若用户在 `time.sleep(10)` 期间按【Ctrl+C】键，则会引发 `KeyboardInterrupt`，若在 `input()` 等待用户输入期间按【Ctrl+Z】键，则会引发 `EOFError`。在下例中处理这些可能的情况：

```

import time

try:
    time.sleep(10) # 仿真一个耗时流程
    num = int(input('输入整数: '))
    print('{0} 为 {1}'.format(num, '奇数' if num % 2 else '偶数'))
except ValueError:
    print('请输入阿拉伯数字')
except (EOFError, KeyboardInterrupt):
    print('用户中断程序')
except:
    print('其他程序例外')

```

当程序中发生例外时，程序执行流程会从例外发生处中断，并进行 `except` 的对比，如果有相符的例外类型，就会执行对应的 `except` 区块，执行完毕后若仍有后续的程序执行流程，就会继续执行。例如：



exceptions average3.py

```

total = 0
count = 0

while True:
    number_str = ''
    try:
        number_str = input('输入数字 (0 结束): ')
        number = int(number_str)
        if number == 0:
            break
        else:
            total += number
            count += 1
    except ValueError as err:
        print('输入的不是整数', number_str)

print('平均值', total / count)

```

在这个范例中，如果输入了非整数的字符串，就会引发 `ValueError` 错误，在执行对应的 `except`

区块后流程继续，由于仍在 `while` 循环中，因此用户仍可进行下一个输入。范例的执行结果如下：

```
输入数字 (0 结束): 10
输入数字 (0 结束): 20
输入数字 (0 结束): 30
输入的不是整数 30
输入数字 (0 结束): 30
输入数字 (0 结束): 40
输入数字 (0 结束): 0
平均值 25.0
```

如果没有相符的例外类型，或者例外完全没有使用 `try..except` 处理，那么例外就会持续往上层调用者传递，在每一层调用处中断，直到最顶层由 `python` 解释器来处理，于是会显示本节一开始看到的 `Traceback` 信息。

7.1.2 例外继承结构

在使用多个 `except` 的时候必须留意一下例外继承结构。如果一个例外在 `except` 的对比过程中符合了某个例外的父类型，后续即使定义了 `except` 对比子类型例外，也等同于没有定义。例如：

```
try:
    dividend = int(input('输入被除数: '))
    divisor = int(input('输入除数: '))
    print('{} / {} = {}'.format(dividend, divisor, dividend / divisor))
except ArithmeticError:
    print('运算错误')
except ZeroDivisionError:
    print('除零错误')
```

执行上面这个程序片段，我们永远不会看到'除零错误'的信息，因为在例外继承结构中，`ArithmeticError` 是 `ZeroDivisionError` 的父类。发生 `ZeroDivisionError` 时，程序在 `except` 对比时会先遇到 `ArithmeticError`，就语义上 `ZeroDivisionError` 是一种 `ArithmeticError`，因此就执行了对应的区块，后续的 `except` 就不会再进行对比了。

在 `Python` 中，例外都是 `BaseException` 的子类，当使用 `except` 而没有指定例外类型时，实际上就是对比 `BaseException`。例如：

```
while True:
    try:
        print('跑跑跑...')
    except:
        print('Shit happens!')
```

上面这个程序无法通过按【`Ctrl+C`】键中断循环，因为只写了 `except` 而没有指定例外类型。这等同于对比了 `BaseException`，也就是全部的例外都会对比成功，包括 `KeyboardInterrupt` 例外，执行过 `except` 区块后，仍在循环之中，因此永不停止。

提示 >>>

如果在文本模式中直接执行了上面的程序，就直接关掉文本模式窗口来结束程序吧！

然而，如果在 `except` 旁边指定了 `Exception` 类型，那么就可以通过按【`Ctrl+C`】键来中断程序。例如：


```

while True:
    try:
        print('跑跑跑...')
    except Exception:
        print('Shit happens!')

```

这是因为 `KeyboardInterrupt` 例外并不是 `Exception` 的子类，因此没有对应的 `except` 可以处理 `KeyboardInterrupt` 例外，因此循环的流程会被中断，最后整个程序结束。

在 Python 标准链接库中，完整的例外继承结构可以在官方文档“Built-in Exceptions¹”中找到，为了便于读者查阅，下面直接列出。

```

BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
|   +-- StopIteration
|   +-- StopAsyncIteration
|   +-- ArithmeticError
|   |   +-- FloatingPointError
|   |   +-- OverflowError
|   |   +-- ZeroDivisionError
|   +-- AssertionError
|   +-- AttributeError
|   +-- BufferError
|   +-- EOFError
|   +-- ImportError
|   +-- LookupError
|   |   +-- IndexError
|   |   +-- KeyError
|   +-- MemoryError
|   +-- NameError
|   |   +-- UnboundLocalError
|   +-- OSError
|   |   +-- BlockingIOError
|   |   +-- ChildProcessError
|   |   +-- ConnectionError
|   |   |   +-- BrokenPipeError
|   |   |   +-- ConnectionAbortedError
|   |   |   +-- ConnectionRefusedError
|   |   |   +-- ConnectionResetError
|   |   +-- FileExistsError
|   |   +-- FileNotFoundError
|   |   +-- InterruptedError
|   |   +-- IsADirectoryError
|   |   +-- NotADirectoryError
|   |   +-- PermissionError
|   |   +-- ProcessLookupError
|   |   +-- TimeoutError
|   +-- ReferenceError
|   +-- RuntimeError
|   |   +-- NotImplementedError
|   |   +-- RecursionError
|   +-- SyntaxError

```

¹ Built-in Exceptions: docs.python.org/3.5/library/exceptions.html

```

|   +-- IndentationError
|       +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|   +-- UnicodeError
|       +-- UnicodeDecodeError
|       +-- UnicodeEncodeError
|       +-- UnicodeTranslateError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning

```

先前谈过，Python 中的例外并非都是错误，`StopIteration` 是一个例子，只是通知迭代的流程无法再进行了；刚才看到的 `KeyboardInterrupt` 也是，表示发生了一个键盘中断；`SystemExit` 则是由 `sys.exit()` 引发的例外，表示离开 Python 程序；`GeneratorExit` 会在生成器的 `close()` 方法被调用时从当时暂停的位置引发，如果在定义生成器时想要在 `close()` 时为生成器做些资源善后等操作，就可以使用。例如：

```

>>> def natural():
...     n = 0
...     try:
...         while True:
...             n += 1
...             yield n
...     except GeneratorExit:
...         print('GeneratorExit', n)
...
>>> n = natural()
>>> next(n)
1
>>> n.close()
GeneratorExit 1
>>>

```

`SystemExit`、`KeyboardInterrupt`、`GeneratorExit` 都直接继承了 `BaseException`，这是因为它们在 Python 中都属于退出系统的例外。如果想要自定义例外，不要直接继承 `BaseException`，而应该继承 `Exception`，或者是 `Exception` 的相关子类来继承。

在继承 `Exception` 自定义例外时，如果自定义了 `__init__()`，建议将自定义的 `__init__()` 传入的自变量通过 `super().__init__(arg1, arg2, ...)` 来调用 `Exception` 的 `__init__()`，因为 `Exception` 的 `__init__()` 默认接受所有传入的自变量，而这些被接受的全部自变量可通过 `args` 属性以一个元组获取。

7.1.3 引发 (raise) 例外

在 5.2.2 小节的 `bank.py` 中曾经创建过一个 `Account` 类，为了讨论方便，这里再将程序代码列出：

```

class Account:
    def __init__(self, name, number, balance):
        self.name = name
        self.number = number
        self.balance = balance

    def deposit(self, amount):
        if amount <= 0:
            print('存款金额不得为负')
        else:
            self.balance += amount

    def withdraw(self, amount):
        if amount > self.balance:
            print('余额不足')
        else:
            self.balance -= amount

    def __str__(self):
        return "Account('{name}', '{number}', {balance})".format(
            name = self.name, number = self.number, balance = self.balance
        )

```

这个 Account 类有什么问题呢？如粗体字部分显示，当存款金额为负或余额不足时，直接在程序流程中使用 print() 显示信息，如果这个 Account 实际上不是运行于文本模式，而是运行在 Web 应用程序或者其他环境呢？print() 的信息将不会出现在这类环境的互动界面上。

可以从另一方面来想，当存款金额为负时，会使得存款流程无法继续而必须中断。类似地，余额不足时，会使得提款流程无法继续而必须中断，如果想让调用方知道因为某些原因而使得流程无法继续而必须中断时，可以引发例外。

在 Python 中如果想要引发例外，可以使用 raise，之后指定要引发的例外对象或类型，只指定例外类型的时候会自动创建例外对象。例如：



exceptions bank.py

```

class Account:
    def __init__(self, name, number, balance):
        self.name = name
        self.number = number
        self.balance = balance

    def check_amount(self, amount):
        if amount <= 0:
            raise ValueError('金额必须是正数:' + str(amount))

    def deposit(self, amount):
        self.check_amount(amount)
        self.balance += amount

    def withdraw(self, amount):
        self.check_amount(amount)

        if amount > self.balance:
            raise BankingException('余额不足')
        self.balance -= amount

```

① 参数值的错误可引发 ValueError

② 检查参数值

③ 商业规则中断可引发自定义例外


```

def __str__(self):
    return "Account('{name}', '{number}', {balance})".format(
        name = self.name, number = self.number, balance = self.balance
    )

class BankingException(Exception):  ← ④ 自定义业务相关的例外
    def __init__(self, message):
        super().__init__(message)

```

在这里定义了一个 `check_amount()` 方法用来检查传入的金额是否为负，因为 `deposit()` 和 `withdraw()` 不应该接受负数，传入负数会是个错误。自变量方面的错误可以引发内建的 `ValueError` ① 错误。对于 `deposit()` 和 `withdraw()`，一开始都会使用 `check_amount()` 方法进行检查 ②。

余额不足属于银行业务流程相关的问题，这部分建议自定义例外 `BankingException` ④，而 `withdraw()` 在余额不足时引发此例外 ③。可以为自己的 API 创建一个根例外，业务相关的例外都可以衍生自这个根例外，这样便于使用 API 的用户在 `except` 时使用根例外来处理 API 相关的例外。

现在 `Account` 的用户若传入负数给 `deposit()` 和 `withdraw()`，或者是提款时余额不足，都会引发例外，只是在发现例外时该怎么处理呢？

传入负数给 `deposit()` 和 `withdraw()` 而引发的 `ValueError` 例外基本上不应该发生，因为正常来说，不应该在存款或提款时输入负数，在设计用户输入界面时，本来就应该有这种“防呆”或防恶意的设计考虑。

如果用户界面上没有此设计，使得 `deposit()` 和 `withdraw()` 真的被输入负数而引发例外，比较好的方式就是不处理例外，让例外浮现到用户界面，看是要在用户界面处理例外还是检查用户输入，总之就是要让用户知道他们自己“做了不该做的事情”。

如果真的要底层调用 `deposit()` 和 `withdraw()` 时处理 `ValueError` 例外，像留下日志信息，那么可以考虑以下类似的流程：

```

try:
    acct.deposit(-500)
except ValueError as err:
    import logging, datetime
    logging.getLogger(__name__).log(
        logging.ERROR,
        'Logging: {time}, {number}, {message}'.format(
            time = datetime.datetime.now(),
            number = acct.number,
            message = err
        )
    )
raise

```

由于例外并没有真的被解决，只是留下了一些日志信息，问题还是要向上传递，因此最后又直接使用 `raise`，这会将 `except` 对比到的例外实例重新引发，就这里的例子来说，会看到类似以下的信息：

```

Logging: 2016-04-20 10:35:15.326126, 123-4567, 金额必须是正数:-500
Traceback (most recent call last):
  File "C:/workspace/exceptions/bank_demo.py", line 5, in <module>
    acct.deposit(-500)
  File "C:/workspace/exceptions/bank.py", line 12, in deposit

```

```

self.check_amount(amount)
File "C:\workspace\exceptions\bank.py", line 9, in check_amount
    raise ValueError('金额必须是正数:' + str(amount))
ValueError: 金额必须是正数:-500

```

若重新引发例外时，想要使用自定义的例外或其他例外类型，并且将 **except** 对比到的例外作为来源，可以使用 **raise from**。例如：

```

try:
    acct.deposit(-500)
except ValueError as err:
    ...
    raise bank.BankingException('输入金额为负的行为已记录') from err

```

在这里，新创建的 **BankingException** 会包含 **ValueError**，因此我们会看到类似以下的信息：

```

Logging: 2016-04-20 10:38:48.350788, 123-4567, 金额必须是正数:-500
Traceback (most recent call last):
  File "C:/workspace/exceptions/bank_demo.py", line 5, in <module>
    acct.deposit(-500)
  File "C:/workspace/exceptions/bank.py", line 12, in deposit
    self.check_amount(amount)
  File "C:/workspace/exceptions/bank.py", line 9, in check_amount
    raise ValueError('金额必须是正数:' + str(amount))
ValueError: 金额必须是正数:-500

```

The above exception was the direct cause of the following exception:

```

Traceback (most recent call last):
  File "C:/workspace/exceptions/bank_demo.py", line 16, in <module>
    raise bank.BankingException('输入金额为负的行为已记录') from err
bank.BankingException: 输入金额为负的行为已记录

```

如果进一步使用 **except** 处理重新引发的 **BankingException**，可以通过例外实例的 **__cause__** 来获取 **raise from** 时的来源例外。如果一个例外在 **except** 中被引发，就算没有使用 **raise from**，原来对比到的例外也会自动被设置给被引发例外的 **__context__** 属性。

例如，**IndexError** 是在 **except** 中引发的，在外层的 **try**、**except** 中对比到 **IndexError** 时，也可以通过 **__context__** 知道例外引发时的 **except** 对比到哪个例外。

```

>>> try:
...     try:
...         raise EOFError('XD')
...     except EOFError:
...         raise IndexError('Orz')
... except IndexError as e:
...     print(e.__cause__)
...     print(e.__context__)
...
None
XD
>>>

```

至于 **withdraw()** 时因为余额而引发的 **BankingException** 例外，由于是个业务流程问题，这就看项目的规格书怎么规范了，也许是在余额不足时转而进行借贷流程。例如：

```

try:
    acct.withdraw(2000)
except bank.BankingException as ex:
    print(ex)

```

```
print('你要进行借贷吗?')
# 其他借贷流程
```

7.1.4 Python 例外风格

在 7.1.1 小节谈到，在 Python 中例外并不一定是错误，例如 `SystemExit`、`GeneratorExit`、`KeyboardInterrupt`，或者是 `StopIteration` 等更像是一种事件，代表着流程因为某个原因无法继续而必须中断。

在 7.1.3 小节的几个 `raise` 范例中就可以清楚地看出这点。例如，当检查出金额为负数时，按照业务上的设计，如果不能再继续流程，就会主动引发一个例外，这并不是嫌程序中的错误不够多，而是对调用者尽告知的义务。

因此，对于标准链接库会引发的例外，也可以从开发标准链接库的开发者角度来思考，为什么他想要引发这样的例外？主动让用户知道发生了例外，用户可以有什么好处？如此就可以知道该怎么处理例外，例如留下日志信息、转为其他流程或重新引发例外。

提示 >>>

在“The art of throwing JavaScript errors”¹ 中有个有趣的比喻，在程序代码的特定点规划出失败，总比预测哪里会出现失败更简单，这就像车体框架的设计，会希望撞击发生框架能以一个可预测的方式溃散，如此制造商才能确保乘客的安全。

在其他程序设计语言中常会有个告诫，例外处理就应当用来处理错误，不应该将例外处理当成是程序流程的一部分。然而在 Python 中，就算例外是个错误，只要程序代码能明确表达出意图，也常被当成流程的一部分。

举个例子来说，如果 `import` 的模块不存在，就会引发 `ImportError` 错误。然而因为 `import` 是条语句，可以出现在语句能出现的场合，因此有时候会想看看某个模块能否被 `import`，若模块不存在，则改为 `import` 另一个模块，此时在 Python 中就会如此编写：

```
try:
    import some_module
except ImportError:
    import other_module
```

这样的编写方式基本上已成了 Python 在 `import` 替代模块时的惯例。实际上，同样的需求也可以通过以下的程序片段来完成：

```
import importlib
some_loader = importlib.find_loader('some_module')
if some_loader:
    import some_module
else:
    import other_module
```

如果指定的模块确实存在，那么 `importlib.find_loader()` 返回值就不会是 `None`，因此可以 `import`

¹ The art of throwing JavaScript errors: <https://goo.gl/xvc6Ee>

指定模块而不会引发 `ImportError`。与使用 `try`、`except` 相比，使用 `importlib.find_loader()` 并进行检查的方式就显得啰嗦了，在必须 `import` 的模块替代方案较多时，这样的方式就显得更为复杂。

而且另一方面，使用 `try`、`except` 的版本就程序代码流程的语义来说，也符合编程惯例，也就是“尝试 `import some_module`，如果引发 `ImportError` 就 `import other_module`”。

因此是否使用 `try`、`except` 处理例外，是否重新引发例外等除了考虑当前已知的信息能否妥善处理例外之外，也可以从程序代码是否能彰显意图来考虑。7.1.3 小节最后在处理 `withdraw()` 方法的例外时，程序代码也显示出“试着提款，如果余额不足引发例外，就进行借贷流程”的意图。

提示 >>>

有机会可以翻阅一下 Python 标准链接库的源码，看看其中引发例外以及使用 `try`、`except` 处理例外的部分，试着揣摩其中的情景，了解为何采用这样的编写方式，这将会有很大的收获。

7.1.5 认识堆栈追踪

在 API 多重的调用下，例外发生点可能在某个函数或方法之中，若想得知例外发生的根源，以及多重调用下例外的传递过程，可以使用 `traceback` 模块。这个模块提供了一些方式，模拟了 python 解释器在处理例外堆栈追踪（Stack Trace）时的行为，可在受控制的情况下获取、格式化或显示例外堆栈追踪信息。

使用 `traceback.print_exc()`

查看堆栈追踪最简单的方法就是直接调用 `traceback` 模块的 `print_exc()` 函数。例如：

`exceptions stacktrace_demo.py`

```
def a():
    text = None
    return text.upper()

def b():
    a()

def c():
    b()

try:
    c()
except:
    import traceback
    traceback.print_exc()
```

在这个范例程序中，`c()` 函数调用 `b()` 函数，`b()` 函数调用 `a()` 函数，`a()` 函数中会因 `text` 为 `None`，之后试图调用 `upper()` 而引发 `AttributeError` 错误。假设事先并不知道这个调用的顺序（也许是在使用一个链接库），当例外发生而被对比后，可以调用 `traceback.print_exc()` 显示堆栈追踪。

```
Traceback (most recent call last):
  File "C:/workspace/exceptions/stacktrace_demo.py", line 12, in <module>
    c()
  File "C:/workspace/exceptions/stacktrace_demo.py", line 9, in c
    b()
```

```
File "C:/workspace/exceptions/stacktrace_demo.py", line 6, in b
    a()
File "C:/workspace/exceptions/stacktrace_demo.py", line 3, in a
    return text.upper()
AttributeError: 'NoneType' object has no attribute 'upper'
```

堆栈追踪信息从上而下 `c()`、`b()`、`a()`、`text.upper()` 的调用顺序，以及引发的例外，每个都标明了源码文件名、行数以及函数名称。如果使用 PyCharm IDE，按下行数就会直接打开源码文件并跳至所对应的行数。

`traceback.print_exc()` 还可以指定 `file` 参数，指定一个已打开的文件对象（File object），将堆栈追踪信息输出到文件里。例如 `traceback.print_exc(file=open('traceback.txt','w+'))` 可将堆栈追踪信息写到 `traceback.txt` 中（有关文件处理的说明，第8章还会详加介绍）。

`traceback.print_exc()` 的 `limit` 参数默认是 `None`，也就是不限制堆栈追踪个数，可以指定为正数或负数。指定为正数就显示最后几次的堆栈追踪个数；指定为负数就倒过来，显示最初几次的堆栈追踪个数。`traceback.print_exc()` 的 `chain` 参数默认是 `True`，也就是一并显示 `__cause__`、`__context__` 等串连起来的例外。

如果只想获取堆栈追踪的字符串描述，可以使用 `traceback.format_exc()`，它会返回字符串，只具有 `limit` 与 `chain` 两个参数。

使用 `sys.exc_info()`

实际上，`print_exc()` 是 `print_exception(*sys.exc_info(), limit, file, chain)` 的便捷（shorthand）方法。`sys.exc_info()` 可获取一个元组对象，当中包括了例外的类型、实例以及 `traceback` 对象。例如：

```
>>> import sys
>>> try:
...     raise Exception('Shit happens!')
... except:
...     print(sys.exc_info())
...
(<class 'Exception'>, Exception('Shit happens!'), <traceback object at 0x00F05DC8>)
```

`traceback` 对象代表了调用堆栈中每一个层次的追踪，可以使用 `tb_next` 获取更深一层的调用堆栈。例如：

exceptions traceback_demo.py

```
import sys

def test():
    raise Exception('Shit happens!')

try:
    test()
except:
    type, value, traceback = sys.exc_info()
    print('例外类型: ', type)
    print('例外对象: ', value)

    while traceback:
        print('.....')
        code = traceback.tb_frame.f_code
        print('文件名: ', code.co_filename)
```

```
print('函数或模块名称: ', code.co_name)

traceback = traceback.tb_next
```

`tb_frame` 代表了该层追踪的所有对象信息, `f_code` 可以获取该层的程序代码信息, 例如 `co_name` 可获取函数或模块名称, `co_filename` 则表示该程序代码所在的文件。上面的范例执行结果如下:

```
例外类型: <class 'Exception'>
例外对象: Shit happens!
.....
文件名: C:/workspace/exceptions/traceback_demo.py
函数或模块名称: <module>
.....
文件名: C:/workspace/exceptions/traceback_demo.py
函数或模块名称: test
```

也可以通过 `tb_frame` 的 `f_locals` 和 `f_globals` 来获取执行时的局部或全局变量, 返回的是个字典对象。

提示 >>>

如果手边已经有了个 `traceback` 对象, 也可以通过 `traceback.print_tb()` 来显示, 或者通过 `traceback.format_tb()` 来获取一个描述字符串, 更多 `traceback` 模块的使用方式可以参考官方帮助文档:

docs.python.org/3.5/library/traceback.html

● 使用 `sys.excepthook()`

对于一个未匹配到的例外, `python` 解释器最后会调用 `sys.excepthook()` 并传入三个自变量: 例外类、实例与 `traceback` 对象, 也就是 `sys.exc_info()` 返回的元组中的三个对象, 默认的操作是显示相关的例外追踪信息 (也就是程序结束前看到的那些信息)。

如果想要自定义 `sys.excepthook()` 被调用时的行为 (或操作), 也可以自行指定一个可接受三个自变量的函数给 `sys.excepthook`, 如果希望对比程序中没有被对比到的其他全部例外, 就可以使用这个特性, 而不一定要在程序最顶层使用 `try`、`except`。例如:

exceptions excepthook_demo.py

```
import sys

def my_excepthook(type, value, traceback):
    print('例外类型: ', type)
    print('例外对象: ', value)

    while traceback:
        print('.....')
        code = traceback.tb_frame.f_code
        print('文件名: ', code.co_filename)
        print('函数或模块名称: ', code.co_name)

        traceback = traceback.tb_next

sys.excepthook = my_excepthook

def test():
```



```
raise Exception('Shit happens!')

test()
```

这个程序的执行结果与上一个范例相同，不过这次采取的是注册 `sys.excepthook` 的方式。

7.1.6 提出警告信息

在 7.1.2 小节谈到例外继承结构，其中 `Exception` 有个子类 `Warning`，当中包括了一些代表着警告信息的子类：

```
BaseException
+-- Exception
    +-- Warning
        +-- DeprecationWarning
        +-- PendingDeprecationWarning
        +-- RuntimeWarning
        +-- SyntaxWarning
        +-- UserWarning
        +-- FutureWarning
        +-- ImportError
        +-- UnicodeWarning
        +-- BytesWarning
        +-- ResourceWarning
```

警告信息通常作为一种提示，用来告知程序有一些潜在性的问题，例如使用了被弃用（`Deprecated`）的功能、以不适当的方式存取资源等。`Warning` 虽然是一种例外，不过基本上不会直接通过 `raise` 引发，而是通过 `warnings` 模块的 `warn()` 函数来提出警告。例如想要提出已弃用的警告，可以如下编写：

```
import warnings
warnings.warn('orz 方法已弃用', DeprecationWarning)
```

在默认的情况下，执行 `warnings.warn()` 函数不会产生任何结果，若想让 `warnings.warn()` 函数起作用，方式之一是在执行 `python` 解释器时通过 `-W` 自变量指定警告控制。例如，如果总是显示警告信息，可以指定 `always`。

```
>python -W always
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import warnings
>>> warnings.warn('orz 方法已弃用', DeprecationWarning)
__main__:1: DeprecationWarning: orz 方法已弃用
>>>
```

`-W` 接受的格式是 `action:message:category:module:lineno`，`always` 是 `action` 的指定值，可指定的值如表 7.1 所示。

表 7.1 警告信息的指定操作

值	说明
error	将警告信息转为例外（引发）
ignore	不显示警告信息
always	总是显示警告信息

(续表)

值	说明
default	只显示每个位置第一个符合的警告信息
module	只显示每个模块第一个符合的警告信息
once	只显示第一个符合的警告信息（无论位置为何）

message 是个正则表达式 (Regular expression)，用来对比想显示的警告消息正文，category 可指定 Warning 的任一子类，默认会是 UserWarning；module 是个正则表达式，用来对比想显示警告信息的模块名称；lineno 是个整数，指定发出警告信息的程序代码对应的行号。

为了了解如何指定警告信息的控制，假设有以下的程序：

exceptions warnings_demo.py

```
import warnings
warnings.warn('orz 方法已弃用', DeprecationWarning)
warnings.warn('XD 用户获得的授权不够', UserWarning)
```

下面是几个警告信息控制的范例。

```
>python warnings_demo.py
warnings_demo.py:3: UserWarning: XD 用户获得的授权不够
  warnings.warn('XD 用户获得的授权不够', UserWarning)

>python -W error:orz warnings_demo.py
Traceback (most recent call last):
  File "warnings_demo.py", line 2, in <module>
    warnings.warn('orz 方法已弃用', DeprecationWarning)
DeprecationWarning: orz 方法已弃用

>python -W ignore::UserWarning warnings_demo.py

>python -W always::DeprecationWarning warnings_demo.py
warnings_demo.py:2: DeprecationWarning: orz 方法已弃用
  warnings.warn('orz 方法已弃用', DeprecationWarning)
warnings_demo.py:3: UserWarning: XD 用户获得的授权不够
  warnings.warn('XD 用户获得的授权不够', UserWarning)

>python -W always::DeprecationWarning::2 warnings_demo.py
warnings_demo.py:2: DeprecationWarning: orz 方法已弃用
  warnings.warn('orz 方法已弃用', DeprecationWarning)
warnings_demo.py:3: UserWarning: XD 用户获得的授权不够
  warnings.warn('XD 用户获得的授权不够', UserWarning)

>python -W always::DeprecationWarning::1 warnings_demo.py
warnings_demo.py:3: UserWarning: XD 用户获得的授权不够
  warnings.warn('XD 用户获得的授权不够', UserWarning)

>
```

如果不想在执行 python 解释器时加上 -W 指定值，也可以设置 PYTHONWARNINGS 环境变量，若已经设置 PYTHONWARNINGS 环境变量，执行时又自行加上了 -W 指定值，则会使用 -W 的指定值。例如：

```
>SET PYTHONWARNINGS=always::DeprecationWarning
```

```
>python warnings_demo.py
warnings_demo.py:2: DeprecationWarning: orz 方法已弃用
  warnings.warn('orz 方法已弃用', DeprecationWarning)
warnings_demo.py:3: UserWarning: XD 用户获得的授权不够
  warnings.warn('XD 用户获得的授权不够', UserWarning)

>python -W error warnings_demo.py
Traceback (most recent call last):
  File "warnings_demo.py", line 2, in <module>
    warnings.warn('orz 方法已弃用', DeprecationWarning)
DeprecationWarning: orz 方法已弃用

>
```

也可以在程序中设置警告信息的控制，例如简单地使用 `warnings.simplefilter()` 方法。

```
>python -W error
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import warnings
>>> warnings.warn('Orz', UserWarning)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UserWarning: Orz
>>> warnings.simplefilter('ignore')
>>> warnings.warn('Orz', UserWarning)
>>> warnings.simplefilter('always')
>>> warnings.warn('Orz', UserWarning)
__main__:1: UserWarning: Orz
>>>
```

`warnings` 模块中还提供了 `filterwarnings()`、`resetwarnings()` 等函数，详细信息可参考 `warnings` 模块的官方文档说明¹。

7.2 例外与资源管理

程序中因错误而抛出例外时，原来的执行流程就会中断，抛出例外处后面的程序代码就不会被执行，如果程序设置了相关资源，使用完毕后是否考虑要关闭资源呢？若因错误而抛出例外，这样的设计是否还能正确地关闭资源呢？

7.2.1 使用 `else`、`finally`

`try`、`except` 的语句其实还可以搭配 `else`、`finally` 来使用。当 `else` 区块出现时，如果 `try` 区块中没有发生例外，`else` 才会执行，如果 `finally` 区块出现，无论 `try` 区块中有没有发生例外，`finally` 区块都一定会执行。

try、except、else

`else` 可与 `try`、`except` 搭配，是其他程序设计语言中不常见的。乍一看 `else`、`finally` 的功能类似，

¹ warnings — Warning control: docs.python.org/3.5/library/warnings.html

不过 `else` 可与 `try`、`except` 搭配，原因在于让 `try` 中的程序代码尽量与可能引发例外的来源相关。例如，在 7.1.1 小节中有个 `average2.py`，其中与引发 `ValueError` 相关的其实是 `int()` 函数的调用，若改以 `try`、`except`、`else`，可以如下编写：



clean-up average.py

```
numbers = input('输入数字 (用空格隔开): ').split(' ')
try:
    ints = [int(number) for number in numbers]
except ValueError as err:
    print(err)
else:
    print('平均值', sum(ints) / len(ints))
```

在这个范例中，`try` 区块在尝试执行 `int()`，紧接着的 `except` 用以对比 `ValueError`。从程序代码上可以清楚地看出，`int()` 与 `ValueError` 的关系如果没有引发例外，就会执行 `else` 区块以显示结果。

在 Python 官方文档“Errors and Exceptions¹”中也有个范例：

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError: # Python 3.3 之后应该是 OSError
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

提示 >>>

从 Python 3.3 开始，`IOError` 已经并入 `OSError`，因此上面的范例程序，`IOError` 应该改成 `OSError`。

在上面的范例中，`open()` 调用时如果没有因文件打开失败而引发例外，就会执行 `else` 区块的内容，这会比编写以下的程序更好。

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
    except IOError: # Python 3.3 之后应该是 OSError
        print('cannot open', arg)
```

在上面的程序中，如果真的引发了例外，那到底是 `open()` 引发的例外还是 `readlines()` 引发的例外呢？如果是 `readlines()` 引发的例外，那么 `except` 中 'cannot open' 的信息显示可能就误导了调试的方向。

try、finally

在 Python 官方文档“Errors and Exceptions”的范例中，实际上 `readlines()` 也是有可能引发例外

¹ Errors and Exceptions: docs.python.org/3.5/tutorial/errors.html

的。如果文件顺利打开，然而 `readlines()` 引发了例外，那么最后的 `f.close()` 就不会被执行，如果想确保 `f.close()` 一定会被执行，可以修改如下：

clean-up read_files.py

```
import sys

for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except FileNotFoundError:
        print('找不到文件', arg)
    else:
        try:
            print(arg, '有', len(f.readlines()), '行')
        finally:
            f.close()
```

由于 `finally` 区块一定会被执行，这个范例要关闭文件的操作，一定要在文件打开成功而 `f` 被指定为文件对象之后，如果这么编写：

```
import sys

for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except FileNotFoundError:
        print('找不到文件', arg)
    else:
        print(arg, '有', len(f.readlines()), '行')
    finally:
        f.close()
```

那么如果文件打开失败，就不会创建 `f` 变量，最后执行 `finally` 的 `f.close()` 时，就会引发 `NameError` 错误并且指出 `f` 名称未定义。

如果程序编写的流程中先 `return` 了，而且也编写了 `finally` 区块，那么 `finally` 区块会先执行完后再将值返回。例如，下面这个范例会先显示“`finally`”再显示“`1`”。

clean-up finally_demo.py

```
def test(flag):
    try:
        if flag:
            return 1
    finally:
        print('finally')
    return 0
```

```
print(test(True))
```

7.2.2 使用 with as

在使用 `try`、`finally` 尝试关闭资源时，经常会发现程序编写的流程是类似的，就如之前 `read_files.py` 示范的，在 `try` 中执行指定的操作，最后在 `finally` 中关闭文件，为了应付之后类似的

需求，可以自定义一个 `with_file()` 函数。例如：

```
def with_file(f, do_it):
    try:
        do_it(f)
    finally:
        f.close()
```

那么 `read_files.py` 就可以运用这个 `with_file()` 函数来改写：

```
import sys

for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except FileNotFoundError:
        print('找不到文件', arg)
    else:
        with_file(f, lambda f: print(arg, '有', len(f.readlines()), '行'))
```

对于其他的需求，也可以重新使用这个 `with_file()` 函数。例如：

```
import sys, logging

def print_each_line(file):
    try:
        # 文件对象可以使用 for in
        # 本书之后会说明
        for line in file:
            print(line, end = '')
    except:
        logger = logging.getLogger(__name__)
        logger.exception('未处理的例外')

try:
    with_file(open(sys.argv[1], 'r'), print_each_line)
except IndexError:
    print('请提供文件名')
    print('范例:')
    print('    python3.5 read.py your_file')
except FileNotFoundError:
    print('找不到文件 {}'.format(sys.argv[1]))
```

实际上，不用自行定义 `with_file()` 这样的函数，Python 提供了 `with as` 语句来满足这类需求。

例如：



clean-up read_files2.py

```
import sys

for arg in sys.argv[1:]:
    try:
        with open(arg, 'r') as f:
            print(arg, '有', len(f.readlines()), '行')
    except FileNotFoundError:
        print('找不到文件', arg)
```

`with` 之后衔接的资源实例可以通过 `as` 来赋值给一个变量，之后就可以在区块中进行资源的处理，如果离开 `with as` 区块，就会自动执行清除资源的操作，这里的例子就是关闭文件。

如果需要同时使用 `with` 来管理多个资源，可以使用逗号 (,) 隔开它们。例如：

```
with open(file_name1, 'r') as f1, open(file_name2, 'r') as f2:
    print(file_name1, '有', len(f1.readlines()), '行')
    print(file_name2, '有', len(f2.readlines()), '行')
```

`with as` 的 `as` 不一定需要。例如：

```
f = open(file_name, 'r')
with f:
    print(file_name, '有', len(f.readlines()), '行')
```

7.2.3 实现上下文管理器

实际上，`with as` 不限使用于文件，只要对象支持上下文管理协议（Context Management Protocol）就可以使用 `with as` 语句。

● 实现 `__enter__()`、`__exit__()`

支持上下文管理协议的对象必须实现 `__enter__()` 与 `__exit__()` 两个方法，这样的对象称为上下文管理器（Context Manager）。

`with` 语句一开始执行就会执行 `__enter__()` 方法，该方法返回的对象可以使用 `as` 赋值给变量（如果有），接着就执行 `with` 区块中的程序代码，下面是个简单的范例。

clean-up context_manager_demo.py

```
class Resource:
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print(self.name, ' __enter__ ')
        return self

    def __exit__(self, type, value, traceback):
        print(self.name, ' __exit__ ')
        return False

with Resource('res') as resource:
    print(resource.name)
```

如果 `with` 区块中的程序代码发生例外，就会执行 `__exit__()` 方法，并传入三个自变量，这三个自变量就是 `sys.exc_info()` 返回的三个对象（参考 7.1.5 小节）。此时 `__exit__()` 方法若返回 `False`，则例外会被重新引发，否则例外就停止传递，通常 `__exit__()` 会返回 `False`，以便在 `with` 之后还可以处理例外。

如果 `with` 区块中没有发生例外而执行完毕，就执行 `__exit__()` 方法，此时 `__exit__()` 的三个参数都接收到 `None`。就上面的例子来说，会如下按序显示：

```
res __enter__
res
res __exit__
```

虽然 `open()` 函数返回的文件对象本身就实现了 `__enter__()` 与 `__exit__()`，不过这里假设它没有，

并自行实现了一个可搭配 with as 的文件读取器，为了进一步了解 open() 函数返回的文件对象，大致可如下实现上下文管理器协议的范例。

clean-up context_manager_demo2.py

```
import sys

class FileReader:
    def __init__(self, filename):
        self.filename = filename

    def __enter__(self):
        self.file = open(self.filename, 'r')
        return self.file

    def __exit__(self, type, msg, traceback):
        self.file.close()
        return False

with FileReader(sys.argv[1]) as f:
    for line in f:
        print(line, end='')
```

使用 @contextmanager

虽然可以直接实现 __enter__()、__exit__() 方法让对象能支持 with as，不过将资源的设置与清除分开到两个方法中实现显得不够清晰明了。可以使用 contextlib 模块的 @contextmanager 来实现，让资源的设置与清除更为清晰明了。例如修改一下刚才的 context_manager_demo2.py:

clean-up context_manager_demo3.py

```
import sys
from contextlib import contextmanager

@contextmanager  ← ①标注@contextmanager
def file_reader(filename):
    try:
        f = open(filename, 'r')
        yield f  ← ②yield 的对象将作为 as 的值
    finally:
        f.close()

with file_reader(sys.argv[1]) as f:
    for line in f:
        print(line, end='')
```

在这里的 file_reader() 函数上标注了 @contextmanager^①，这表示此函数将会返回一个实现了上下文管理器协议的对象，函数中只要按需求编写 try、finally 即可，重点在于设置的资源如果要搭配 with as 的 as 设置值，就要将资源接在 yield 之后^②。

实际上，with as 语句是用来表示其区块处于某个特殊的上下文之中，处于自动关闭文件的上下文（或情景）时只是其中一种情况，因此我们也可以实现一个上下文管理器用于抑制指定的例外。

clean-up context_manager_demo4.py

```
import sys
from contextlib import contextmanager
```

```
@contextmanager
def suppress(ex_type):
    try:
        yield
    except ex_type:
        pass

with suppress(FileNotFoundError):
    for line in open(sys.argv[1]):
        print(line, end='')

```

可以看到，使用@contextmanager 实现函数时 yield 的前后创建了 with 区块的上下文。实际上，contextlib 模块提供了 suppress() 这个函数可供调用。

clean-up context_manager_demo5.py

```
import sys
from contextlib import suppress

with suppress(FileNotFoundError):
    for line in open(sys.argv[1]):
        print(line, end='')

```

如果某个对象实现了 close() 方法，但没有实现上下文管理器协议，仍然有让它搭配 with as 来使用的方式。例如：

clean-up context_manager_demo6.py

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()

class Some:
    def __init__(self, name):
        self.name = name

    def close(self):
        print(self.name, 'is closed.')

with closing(Some('Resource')) as res:
    print(res.name)

```

实际上，contextlib 模块就提供了 closing() 这个函数可供调用，因此上面的范例可以直接改写如下：

clean-up context_manager_demo6.py

```
from contextlib import closing

class Some:
    def __init__(self, name):
        self.name = name

```



```
def close(self):
    print(self.name, 'is closed.')

with closing(Some('Resource')) as res:
    print(res.name)
```

contextlib 模块中还有 `redirect_stdout()`、`redirect_stderr()` 函数可供使用，用于将标准输出或标准错误重新导向到指定目标（例如一个文件）的上下文，更多有关于 contextlib 模块的介绍可以直接参考官方帮助文档：

docs.python.org/3.5/library/contextlib.html

7.3 重点复习

python 解释器尝试执行 try 区块中的程序代码，如果发生例外，执行流程会跳离例外发生点，然后对比 except 声明的类型，看是否符合引发的例外对象类型，如果是，就执行 except 区块中的程序代码。

在 Python 中，例外并不一定是错误，例如使用 for in 语句时，其实底层就运用到了例外处理机制。

只要是具有 `__iter__()` 方法的对象都可以使用 for in 来迭代。迭代器具有 `__next__()` 方法，每次迭代时就会返回下一个对象，没有下一个元素时则会引发 StopIteration 例外。

可以使用 `iter()` 方法调用对象上的 `__iter__()` 来获取迭代器，再使用 `next()` 来调用迭代器的 `__next__()` 方法。

except 之后可以使用元组指定多个对象，也可以有多个 except，如果没有指定 except 后的对象类型，表示捕捉所有引发的对象。

如果一个例外在 except 的对比过程中就符合了某个例外的父类型，后续即使定义了 except 对比子类型的例外也等同于没有定义。

在 Python 中，例外都是 BaseException 的子类，当使用 except 而没有指定例外类型时，实际上就是对比 BaseException。如果想要自定义例外，不要直接继承 BaseException，而应该继承 Exception，或者是 Exception 的相关子类来继承。

在继承 Exception 自定义例外时，如果自定义了 `__init__()`，建议将自定义的 `__init__()` 传入的自变量通过 `super().__init__(arg1, arg2, ...)` 来调用 Exception 的 `__init__()`，因为 Exception 的 `__init__()` 默认接受所有传入的自变量，而这些被接受的全部自变量可通过 args 属性以一个 tuple 来获取。

如果想让调用方知道因为某些原因而使得流程无法继续而必须中断时，可以引发例外。在 Python 中如果想要引发例外，可以使用 `raise`，之后指定要引发的例外对象或类型，只指定例外类型的时候会自动创建例外对象。

可以为自己的 API 创建一个根例外，业务相关的例外都可以衍生自这个根例外，这便于 API 用户在 except 时使用自己的根例外来处理 API 相关的例外。

直接使用 `raise` 会将 except 对比到的例外实例重新引发。若重新引发例外时，想要使用自定义的例外或其他例外类型，并且将 except 对比到的例外作为来源，可以使用 `raise from`。

可以通过例外实例的 `__cause__` 来获取 `raise from` 时的来源例外。如果一个例外在 except 中被引

发, 就算没有使用 `raise from`, 原来对比到的例外也会自动设置给被引发例外的 `__context__` 属性。

在 Python 中, 例外并不一定是错误, 例如 `SystemExit`、`GeneratorExit`、`KeyboardInterrupt` 或 `StopIteration`, 更像是一种事件, 代表着流程因为某个原因无法继续而必须中断。

主动引发一个例外并不是嫌程序中的错误不够多, 而是对调用者尽告知的义务。在 Python 中, 就算例外是个错误, 只要程序代码能明确表达出意图, 也常会当成流程的一部分。

有时候会想看看某个模块能否被 `import`, 若模块不存在, 则改为 `import` 另一个模块, 此时在 Python 中就会如此编写:

```
try:
    import some_module
except ImportError:
    import other_module
```

若想得知例外发生的根源, 以及多重调用下例外的传递过程, 可以使用 `traceback` 模块。

警告信息通常作为一种提示, 用来告知程序有一些潜在性的问题, 例如使用了被弃用 (`Deprecated`) 的功能、以不适当的方式存取资源等。Warning 虽然是一种例外, 不过基本上不会直接通过 `raise` 引发, 而是通过 `warnings` 模块的 `warn()` 函数来发出警告。

`else` 可与 `try`、`except` 搭配的原因在于, 让 `try` 中的程序代码尽量与可能引发例外的来源相关。

如果程序编写的流程中先 `return` 了, 而且也编写了 `finally` 区块, 那么 `finally` 区块会先执行完后, 再将值返回。

`with` 之后衔接的资源实例可以通过 `as` 来赋值给一个变量, 之后就可以在区块中进行资源的处理, 当离开 `with as` 区块之后就会自动执行清除资源的操作。

`with as` 不限使用于文件, 只要对象支持上下文管理协议, 就可以使用 `with as` 语句。

支持上下文管理协议的对象必须实现 `__enter__()` 与 `__exit__()` 两个方法, 这样的对象称为上下文管理器。可以使用 `contextlib` 模块的 `@contextmanager` 来实现, 让资源的设置与清除更为清晰明了。

`with as` 语句是用来表示其区块处于某个特殊的上下文之中, 处于自动关闭文件的上下文只是其中一种情况。

使用 `@contextmanager` 实现函数时, `yield` 的前后创建了 `with` 区块的上下文。

课后练习

实践题

1. 针对 7.1.3 小节设计的 `Account` 类重新设计例外, 在 `deposit()`、`withdraw()` 方法的自变量为负时, 使用指定信息与当时的负数创建 `IllegalMoneyException` 例外并将其引发。在 `withdraw()` 方法余额不足时, 使用指定的信息与当时的余额创建 `InsufficientException` 例外并将其引发。`IllegalMoneyException` 与 `InsufficientException` 都必须继承自 `BankingException`。

2. 请设计一个 `Suppress` 类实现上下文管理器的 `__enter__()`、`__exit__()` 方法, 可指定想要抑制的例外类型。例如:

```
with Suppress(FileNotFoundError):
```

```
for line in open(sys.argv[1]):
    print(line, end='')
```

3. 请实现一个 `contextmanager()` 函数，当执行以下程序时：

```
def suppress(ex_type):
    try:
        yield
    except ex_type:
        pass

suppress = contextmanager(suppress)
with suppress(FileNotFoundError):
    for line in open(sys.argv[1]):
        print(line, end='')
```

`FileNotFoundError` 会被抑制，也就是说，这个练习的 `context manager()` 函数模仿了 `contextlib` 的 `@contextmanager` 功能，因此也可以具有以下功能：

```
class Some:
    def __init__(self, name):
        self.name = name

    def close(self):
        print(self.name, 'is closed.')

def closing(thing):
    try:
        yield thing
    finally:
        thing.close()

closing = contextmanager(closing)
with closing(Some('Resource')) as res:
    print(res.name)
```

这是个选择性的高级练习题，若想挑战一下自己的能力，可参考 `contextlib` 模块源码中 `contextmanager()` 函数的实现。

第 8 章

open()与 io 模块

学习目标

- 使用 open()函数
- 使用 stdin、stdout、stderr
- 认识文件描述符
- 认识 io 模块

```
workspace\pdb_demo>python -m pdb filter_demo2.py
> workspace\pdb_demo\filter_demo2.py (1) <module>
> filter_demo2.py
> filter_demo2.py
> filter_demo2.py
```

```
> def filter_it(predicate, lt):
    <category>: epinome
    filter_it(len_greater_than(5), lt)
    (Pdb)> lt(len_greater_than(5), lt)
```

```
1 -> def filter_it(predicate, lt):
```

```
2     result = []
```

```
3     for elem in lt:
```

```
4         if predicate(elem):
```

```
5             result.append(elem)
```

```
6     return result
```

```
7 Get Packages:
```

```
8 def len_greater_than(num):
```

```
9     def len_greater_than_num(elem):
```

```
10         return len(elem) > num
```

```
11     return len_greater_than_num
```

```
(Pdb)
```



8.1 使用 open()函数

在 Python 中想要进行文件读写，基本上可以从内建的 open()函数使用出发，这是个工厂函数，隐藏了文件对象（File object）的相关细节，使得 open()函数可以应付文件读写的大部分需求。

8.1.1 file 与 mode 参数

如果想使用 open()函数进行文件的读写，对于最基本的需求，只需要使用到它的前两个参数：**file** 与 **mode**。file 可以是个字符串，指定要读写文件的路径，可指定相对路径（相对于当前的路径）或者绝对路径；mode 是使用字符串来指定文件打开的模式，可以指定的字符串及其含义如表 8.1 所示。

表 8.1 文件打开的模式

字符串	说明
r	读取模式（默认）
w	写入模式，会先清空文件内容
x	只在文件不存在时才创建新文件并打开为写入模式，若文件已存在会引发 FileExistsError
a	附加模式，若文件已存在，写入的内容会附加到文件尾端
b	二进制模式
t	文本模式（默认）
+	更新模式（读取与写入）

open()的 mode 默认是'r'，在只指定'r'、'w'、'x'、'a'的情况下就相当于以文本模式打开，也就等同于'rt'、'wt'、'xt'、'at'，如果要以二进制模式打开，就要指定'rb'、'wb'、'xb'、'ab'。

如果想以更新模式打开，对于文本模式，可以使用'r+'、'w+'、'a+'，对于二进制模式，可以使用'r+b'、'w+b'、'a+b'。

read()、write()、close()

下面直接来设计一个通用的 upper 程序，可使用命令行参数指定源与目标，将源文本文件的内容全部转成大写后写入目标文件，这同时示范了文本模式下文件的读取与写入。

basicio upper.py

```
import sys

src_path = sys.argv[1]
dest_path = sys.argv[2]

with open(src_path) as src, open(dest_path, 'w') as dest:
    content = src.read()
```

① 分别以'r'与'w'模式打开

↓

← ② 使用 read()读取数据

```
dest.write(content.upper()) ← ❸使用 write()写入数据
```

这个 `upper` 程序会从命令行参数获取源文件与目标文件的路径，程序中分别以 `'r'`（`mode` 参数默认值）及 `'w'` 模式使用 `open()` 打开源文件与目标文件❶。对于每个被打开的文件，建议在不使用时调用 `open()` 返回的文件对象的 `close()` 方法明确地关闭文件。文件对象实现了上下文管理器协议（详细内容见 7.2.3 小节），因而可使用 `with` 代替我们进行文件关闭的操作。

提示 >>>

实际上，文件对象实现的 `__del__()` 方法中会调用 `close()` 方法，因此当文件对象没有任何名称引用而将被回收前也会自动关闭文件。不过，由于资源回收的时机无法预测，因此建议自行明确地关闭文件。

`read()` 方法在未指定自变量的情况下会读取文件全部的内容。对于文本模式来说，会返回 `str` 实例❷（对于二进制文件来说，会返回 `bytes` 实例），因此范例中可以使用 `upper()` 将其中的字符全部转为大写。`write()` 方法可将指定的数据写入文件。对于文本模式来说，`write()` 接受 `str` 实例❸并返回写入的字符数（对于二进制文件来说，`write()` 接受 `bytes` 实例并返回写入的字节数）。

`read()` 方法在指定整数自变量的情况下会读取指定的字符数或字节数（视打开模式是文字或者二进制而定）。

文件打开模式与后续进行的操作必须符合，否则会引发 `Unsupported Operation` 例外。例如使用 `'r'` 模式打开却要进行写入的操作。可以使用 `readable()` 方法测试是否可读取，使用 `writable()` 方法测试是否可写入。

```
>>> f = open('upper.py')
>>> f.write('write it?')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
io.UnsupportedOperation: not writable
>>> f.readable()
True
>>> f.writable()
False
>>> f.close()
>>>
```

❹ `readline()`、`readlines()`、`writelines()`

有趣的是，无论是文本模式还是二进制模式都可以使用 `readline()`、`readlines()`、`writelines()` 方法。

对于文本模式来说，默认读取到 `'\n'`、`'\r'` 或 `'\r\n'` 时都可以被判定为一行，而 `readline()` 或 `readlines()` 读到的每一行换行字符都一律换为 `'\n'`。对于二进制模式来说，行的判断标准默认是遇到 `b'\n'` 这个 `bytes` 类型。

文本模式在写入的情况下，任何 `'\n'` 都会被置换为 `os.linesep` 的值（Windows 就是 `'\r\n'`）。

下面的范例故意以 `'rb'` 模式打开 `.py` 文件，看看 `readlines()` 会返回什么样的内容：

```
>>> f = open('upper.py', 'rb')
>>> f.readlines()
```



```
[b'import sys\r\n', b'\r\n', b'src_path = sys.argv[1]\r\n', b'dest_path = sys.argv[2]\r\n',
b'\r\n', b"with open(src_path) as src, open(dest_path, 'w') as dest:\r\n", b'    content =
src.read()\r\n', b'    dest.write(content.upper())']
>>> f.close()
>>>
```

可以看到，`readlines()`方法会以 `list` 返回文件的内容，`list` 中每个元素是文件中被判定为一行的内容。

如果想逐行读取文件内容，基本上使用 `readline()`方法搭配 `while` 循环即可，`readline()`在读不到下一行时会返回空字符串，因此可以这么编写：

```
>>> with open('upper.py') as f:
...     while True:
...         line = f.readline()
...         if not line:
...             break
...         print(line, end = '')
...
import sys

src_path = sys.argv[1]
dest_path = sys.argv[2]
```

不过，`open()`返回的文件对象都实现了 `__iter__()`方法，可以返回一个迭代器，因此可以直接使用 `for in` 来进行迭代，每次迭代都相当于执行 `readline()`，所以上面的例子可以改写为：

```
>>> with open('upper.py') as f:
...     for line in f:
...         print(line, end = '')
...
import sys

src_path = sys.argv[1]
dest_path = sys.argv[2]
```

可以看到，这样的写法简洁多了，这正是 **Python** 的文件读取风格：读取一个文件最好的方式就是不要去 `read`。

🕒 `tell()`、`seek()`、`flush()`

在进行文件读写时，`tell()`方法可以告知文件指针当前在文件中的位移量，单位是字节数，文件开头的位移量是 `0`，`seek()`方法可以指定跳到哪个位移量。为了示范，接下来假设有个 `test.txt` 文件中输入了 `12345` 并存盘。

```
>>> f = open('test.txt', 'rb')
>>> f.tell()
0
>>> f.read(1)
b'1'
>>> f.read(1)
b'2'
>>> f.tell()
2
>>> f.seek(1)
```

```

1
>>> f.read(1)
b'2'
>>> f.close()
>>>

```

如果是在 Windows 中创建纯文本文件，那么应该采取 MS936 编码，12345 每个字符会占一个字节，采用二进制打开模式每次 `read(1)` 会读取一个字节，在上面的范例中就可以看到相应的结果。实际上，`seek()` 的返回值是操作之后文件指针在文件中的位移量。

因此，可以使用 `seek()` 来实现文件的随机存取，例如将第三个字节改为 `b'0'`。

```

>>> f = open('test.txt', 'r+b')
>>> f.seek(2)
2
>>> f.write(b'0')
1
>>> f.flush()
>>> f.seek(0)
0
>>> f.read()
b'12045'
>>> f.close()
>>>

```

这次的打开模式是 `'r+b'`，表示为可读取与更新模式，请不要写成 `'w+b'`，这样会清空文件内容，也不要写成 `'a+b'`，这样会将写入的数据放到文件尾端。由于文件对象默认会缓冲处理，不一定会马上看到文件中写入了数据，这时可以执行 `flush()` 方法，将缓冲内容清空并写入文件。

► readinto()

在二进制模式中，`read()` 方法返回的 `bytes` 是不可变动的，如果想将读取到的字节数据收集到一个列表 (list) 中，基本上可以使用 `list()`，将 `read()` 到的 `bytes` 转为列表 (list)，例如：

```

>>> content = None
>>> with open('test.txt', 'rb') as f:
...     content = f.read()
...
>>> list(content)
[49, 50, 51, 52, 53]
>>>

```

不过，二进制模式中的文件对象拥有一个 `readinto()` 方法接受 `bytearray` 实例，可以直接将读取到的数据传入，这样就可以不用中介的变量。例如：

```

>>> import os.path
>>> b_arr = bytearray(os.path.getsize('test.txt'))
>>> with open('test.txt', 'rb') as f:
...     f.readinto(b_arr)
...
5
>>> b_arr[0]
49
>>> b_arr[1]
50
>>> b_arr
bytearray(b'12345')
>>>

```

注意>>>

无论是通过 `list()` 将 `bytes` 转为列表 (`list`) 实例, 还是使用 `bytearray`, 通过索引获取的每个元素都是 `int` 类型, 可以通过 `bytes([value])` 将某个整数值转为 `bytes`, 例如 `bytes([49])` 结果会是 `b'1'`。

8.1.2 buffering、encoding、errors、newlines 参数

`open()` 函数实际上有 8 个参数, 刚才说明的 `file` 与 `mode` 是最常使用的参数, 接下来我们再看看 `buffering`、`encoding`、`errors` 与 `newlines` 参数, 至于 `closed`、`opener` 参数的说明将放在 8.2.1 小节中。

buffering

这个参数用来设置缓冲策略, 默认的缓冲策略会试着自行决定缓冲区大小 (通常会 4096 字节或 8192 字节), 或者对互动文本文件 (`isatty()` 为 `True` 时, 例如 Windows 的命令提示符) 采用行缓冲 (`line buffering`)。

`buffering` 设置为 0 表示关闭缓冲, 设置为大于 0 的整数值表示指定缓冲区的字节大小。

举个例子来说, 在 8.1.1 小节看到的随机存取范例, 如果采用 `f=open('test.txt', 'r+b', buffering = 0)`, 在 `f.write()` 更新文件之后不必使用 `f.flush()`, 马上就可以看到文件内容的变化。

encoding 与 errors

指定文本模式时默认采用 `locale.getpreferredencoding()` 的返回值作为文件编码, 以 Windows 简体中文版来说是返回 `'cp936'` 或 `'gbk'`。

提示>>>

在 Python 执行环境中, 将 `MS936`、`CP936` 和 `gbk` 视为同一套编码。

如果文本文件采用的编码与 `locale.getpreferredencoding()` 的返回值不同, 读取时就有可能出现乱码问题, 例如有个文件中编写了中文, 以文本模式打开且未指定 `encoding` (编码), 在读取中文时会试着一次读取两个字节, 若有个 `test_ch.txt` 是以 UTF-8 编码并且编写了中文, 那么读取时就会出现乱码。例如:

```
>>> with open('test_ch.txt', 'r') as f:
...     print(f.read(1))
...     print(f.tell())
...
娴
2
>>>
```

在上面的例子中, 假设 `test_ch.txt` 中写了“测试”两个汉字, 由于 UTF-8 对这两个汉字的编码是每个汉字三个字节, 因此只读取两个字节时就会出现乱码。在正确指定 `encoding` 为 `'UTF-8'` 后就不会有问题了。

```
>>> with open('test_ch.txt', 'r', encoding = 'UTF-8') as f:
```



```
...     print(f.read(1))
...     print(f.tell())
...
测
3
>>>
```

`errors` 参数可指定发生编码错误时该如何进行处理？在不设置的情况下，发生编码错误时会引发 `ValueError` 的子类例外，例如 `test_ch2.txt` 文件中有“方法已弃用”这几个汉字，并采用 UTF-8 编码，若如下读取，将会引发 `UnicodeDecodeError` 错误。

```
>>> with open('test_ch2.txt') as f:
...     print(f.read())
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
UnicodeDecodeError: 'gbk' codec can't decode byte 0xa8 in position 14: illegal multibyte sequence
>>>
```

如果设置 `errors` 为 `'ignore'`，那么会忽略错误，继续进行读取的操作，结果如下所示。

```
>>> with open('test_ch2.txt', errors = 'ignore') as f:
...     print(f.read())
...
鑑規磬宸插純鑾
>>>
```

`errors` 的其他可设置选项可参考 `open()` 函数¹ 的说明。

newlines

8.1.1 小节曾介绍过 `newlines` 参数，对于文本模式来说，默认读取到 `'\n'`、`'\r'` 或 `'\r\n'` 都可以被判定为一行，而 `readline()` 或 `readlines()` 读到的每一行换行字符都一律换为 `'\n'`。

`newlines` 的指定值还可以是 `"`、`'\n'`、`'\r'` 与 `'\r\n'`。如果指定了 `"`，读取到 `'\n'`、`'\r'` 或 `'\r\n'` 都可以被判定为一行，而 `readline()` 或 `readlines()` 读到的每一行一律保留来源的换行字符，如果设置为其他 `'\n'`、`'\r'` 或 `'\r\n'`，那么读取后的换行字符就会是指定的字符。

文本模式在写入的默认情况下，任何 `'\n'` 都会被置换为 `os.linesep` 的值。如果 `newline` 设为 `"` 就保留原有的换行字符，如果指定为其他值，就以指定的字符进行置换。

8.1.3 stdin、stdout、stderr

目前获取用户的输入都是使用 `input()` 函数，若想显示指定的值，则使用 `print()` 函数，它们各自会使用预先连接的设备进行输入或输出。预先连接的输入、输出设备被称为标准输入（**Standard input**）与标准输出（**Standard output**），以个人计算机而言，通常对应到终端的输入与输出。

在 `sys` 模块中有个 `stdin` 代表标准输入，`stdout` 代表标准输出，它们的操作就像 `open()` 函数打开的文本模式的文件对象。例如，下面的范例模仿了 `input()` 函数的实现。

¹ `open()` 函数：docs.python.org/3.5/library/functions.html#open

basicio stdin_demo.py

```
import sys

def console_input(prompt):
    sys.stdout.write(prompt)  ← ❶ 使用标准输出
    sys.stdout.flush()      ← ❷ 从缓存清出数据到标准输出
    return sys.stdin.readline() ← ❸ 使用标准输入读取一行

name = console_input('请输入名称: ')
print('哈啰, ', name)
```

我们可以使用 `sys.stdout` 的 `write()` 方法“写出”信息❶，为了马上能看到指定的信息显示，必须使用 `flush()` 方法清出数据❷（即将缓冲区中的数据清出到标准输出），接着可以使用 `sys.stdin.readline()` 读入一行输入的信息❸。

提示 >>>

对于 Windows 的命令提示符来说，文字编码默认会使用命令提示符的代码页（codepage），其他平台则使用 `locale.getpreferredencoding()`，也可以自行使用环境变量 `PYTHONIOENCODING` 来设置，设置方式可详见：

docs.python.org/3.5/using/cmdline.html#envvar-PYTHONIOENCODING

对于标准输入或输出，若想要以二进制模式读取或写入，可以使用 `sys.stdin.buffer` 或 `sys.stdout.buffer`，它们的操作或行为就像以 `open()` 函数打开的二进制模式的文件对象。

实际上可以改变标准输入或输出的来源，例如，将一个以 `open()` 函数打开的文件对象赋值给 `sys.stdin` 就可以利用 `input()` 来读取。例如：

```
>>> import sys
>>> sys.stdin = open('stdin_demo.py', encoding = 'UTF-8')
>>> input()
'import sys'
>>> input()
''
>>> input()
'def console_input(prompt):'
>>> input()
'    sys.stdout.write(prompt)'
>>> input()
'    sys.stdout.flush()'
>>> input()
'    return sys.stdin.readline()'
>>> input()
''
>>>
```

类似地，将一个打开的文件对象赋值给 `sys.stdout` 就可以利用 `print()` 把数据写到文件中。不过内建的 `print()` 函数本身就有个 `file` 参数可以满足这样的需求。

```
>>> with open('data.txt', 'w') as f:
...     print('Hello, World', file = f)
...
>>>
```

上面的程序执行过后，工作目录下就会发现有个 data.txt 文件，内容就是执行 print() 函数时输出的 'Hello, World' 信息。

在文本模式下，可以使用 > 将程序执行时的标准输出信息重定向到指定的文件，或者使用 >> 附加信息。例如：

```
>python -c "print('Hello, World')" > data.txt
>python -c "print('Hello, World')" >> data.txt
>
```

执行以上指令，标准输出的信息被直接重新定向到 data.txt，因此不会看到信息在屏幕上的显示，而 data.txt 中会出现两行 'Hello, World' 文字。实际上还有个 sys.stderr 代表标准错误 (Standard error) 设备，就 Windows 而言，默认也就是命令提示符程序 (即终端程序)，标准错误的输出不能使用 > 或 >> 重新定向至文件。例如：

```
>python -c "import sys; sys.stderr.write('Hello, World')" > data.txt
Hello, World
>python -c "import sys; sys.stderr.write('Hello, World')" >> data.txt
Hello, World
>
```

在上面的范例中，由于使用 sys.stderr 写出信息就不能使用 > 或 >> 重新定向，因此信息仍然在屏幕上显示出来了，而 data.txt 的内容会是一片空白。

提示 >>>

看到了吗？Python 中还可以使用分号 (;)，想要在一行中写两条语句时就可以使用。

8.2 高级文件处理

在处理文件时，使用 open() 函数可以应付绝大多数的情况。然而，open() 函数实际上是个工厂函数 (Factory function)，隐藏了创建文件对象的细节，为了进一步满足需求，或者更清楚地知道当前手上正在操作的文件对象特性为何，认识 open() 函数背后的一些细节仍是必要的，这样也不至于沦落到死记硬背 API 的窘境。

8.2.1 认识文件描述符

open() 函数的 file 参数除了接受字符串指定文件的路径之外，还可以指定文件描述符 (File descriptor)，文件描述符会是个整数，对应到当前程序已打开的文件。举例来说，标准输入通常使用文件描述符 0，标准输出是 1，标准错误对应的文件描述符为 2，进一步打开的文件对应的文件描述符依次是 3、4、5 等数字。

对于文件对象，可以使用 fileno() 方法来获取文件描述符。例如：

```
>>> import sys
>>> sys.stdin.fileno()
0
>>> sys.stdout.fileno()
```



```

1
>>> sys.stderr.fileno()
2
>>> with open('data.txt') as f:
...     print(f.fileno())
...
3
>>>

```

`open()` 的 `file` 参数也可以接受文件描述符，文件描述符的值是个整数，如果指定 `open()` 的第一个参数为 0 会如何呢？

```

>>> f = open(0)
>>> f.readline()
This is a test!
'This is a test!\n'
>>>

```

因为标准输入的文件描述符值是 0，因此上面的例子中，`f.readline()` 会从标准输入读入一行。

实际上在 8.1.3 小节说明了，`sys.stdin` 本身已经有文件对象的行为，不必特别使用 `open()` 来包裹(wrap)，这里只是为了示范。若能获取一个对应到系统上已打开的文件描述符，就有机会使用 `open()` 包裹成为文件对象，进一步使用文件对象的高级操作。

如果想打开一个文件描述符，可以使用 `os` 模块的 `open()` 函数，最基本的使用方式是指定 `path` 与 `flags` 参数。例如想以只读方式打开指定的文件描述符并读取 5 个字节，程序语句如下所示：

```

>>> import os
>>> fd = os.open('test.txt', os.O_RDONLY)
>>> os.read(fd, 5)
b'12345'
>>> os.close(fd)
>>>

```

`os.read()`、`os.close()` 等都属于低级操作，若想了解更多文件描述符的细节，可以参考“File Descriptor Operations¹”。

之所以要提到文件描述符，是因为可以用它来说明 `open()` 函数的 `closed` 参数，它的默认值是 `True`，这表示若 `open()` 时 `file` 指定了一个文件描述符，在文件对象调用 `close()` 方法而关闭时被指定的文件描述符也会一并被关闭，当设置为 `False` 时就不会关闭被指定的文件描述符。

`open()` 还有个 `opener` 函数，可以用来指定一个函数，该函数必须有两个参数，第一个参数会传入 `open()` 函数被指定的文件路径，第二个参数会是 `open()` 函数按 `mode` 计算出来的 `flags` 值，函数最后必须返回一个文件描述符，`open()` 函数基于该文件描述符创建文件对象，以便进行文件操作。

因此，如果想在打开文件时执行一些加工操作，就可以指定 `opener` 参数。下面是一个简单范例，指定的 `opener` 函数可以在文件不存在时以 `sys.stdout` 的文件描述符来替代。

```

>>> import sys
>>> import os
>>> def or_stdout(path, flags):
...     if os.path.exists(path):
...         return os.open(path, flags)
...     else:

```

¹ File Descriptor Operations: docs.python.org/3.5/library/os.html#file-descriptor-operations

```

...     return sys.stdout.fileno()
...
>>> f = open('xyz.txt', 'w', opener = os_stdout)
>>> f.write('Hello, World\n')
Hello, World
13
>>>

```

在上面的范例中，实际上 xyz.txt 并不存在，因此 open() 实际上会使用标准输出，结果就是直接显示 write() 的输出信息。

8.2.2 认识 io 模块

open() 函数实际上是个工厂函数，当根据需求指定某些自变量之后，open() 函数在后台会进行文件的打开、相关文件对象的创建与设置、将文件对象返回等操作。如果想进一步掌握文件对象的操作，就得认识 io 模块，相关的文件对象就定义在此模块中。

可以来看看指定不同的自变量时返回的文件对象会是什么类型。

```

>>> open('test.txt')
<_io.TextIOWrapper name='test.txt' mode='r' encoding='cp936'>
>>> open('test.txt', 'rb')
<_io.BufferedReader name='test.txt'>
>>> open('test.txt', 'rb', buffering = 0)
<_io.FileIO name='test.txt' mode='rb' closefd=True>
>>>

```

实际上，TextIOWrapper、BufferedReader、FileIO 等是在 _io 模块中，并且在 io.py 中定义了相同的名称引用至 TextIOWrapper、BufferedReader、FileIO 等。

文件对象的继承结构

Python 的 I/O 大致上分为三个主要类型：文本 (Text) I/O、二进制 (Binary) I/O 与原始 (Raw) I/O。符合这些分类的具体对象就是之前一直看到的文件对象，而 TextIOWrapper、BufferedReader、FileIO 分别属于这三种类型。

在 io 模块中各类继承方面，IOBase 是所有 I/O 类的基类，作用于字节串流之上，而 TextIOBase、BufferedIOBase、RawIOBase 是 IOBase 的子类，分别代表文本 I/O、二进制 I/O、原始 I/O 的基类。io 模块¹的帮助文档有个表格列出了这几个类的关系与定义的方法，如表 8.2 所示。

表 8.2 I/O 基类与方法定义

基类	继承	抽象方法	Mix in 方法
IOBase		fileno、seek、truncate	close、closed、__enter__、__exit__、flush、isatty、__iter__、__next__、readable、readline、readlines、seekable、tell、writable、writelines

¹ io 模块：docs.python.org/3.5/library/io.html

(续表)

基类	继承	抽象方法	Mix in 方法
RawIOBase	IOBase	readinto、write	继承自 IOBase 的方法，以及 read、readall
BufferedIOBase	IOBase	detach、read、read1、write	继承自 IOBase 的方法，以及 readinto
TextIOBase	IOBase	detach、read、readline、write	继承自 IOBase 的方法，以及 encoding、errors、newlines

刚才看到的 TextIOWrapper、BufferedReader、FileIO 分别是 TextIOBase、BufferedIOBase、RawIOBase 的子类，可操作的方法在 8.1 小节大多说明过了，至于其他子类与继承关系，如图 8-1 所示。

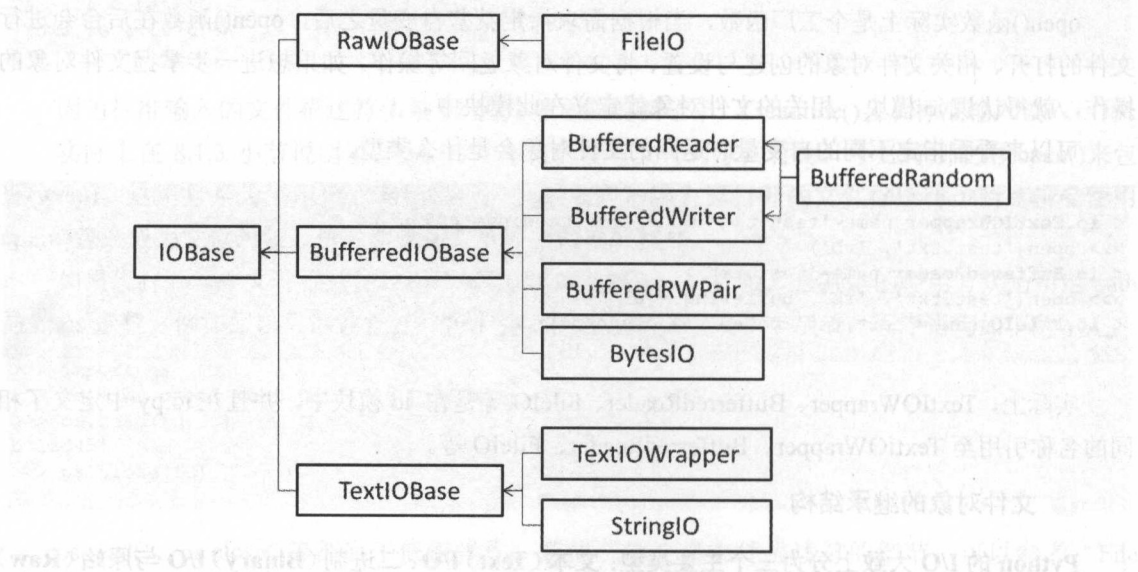


图 8-1 io 模块中类的继承结构

创建文件对象

原始 I/O 是无缓冲的低级操作，很少直接使用，通常作为文本 I/O 或二进制 I/O 的底层操作。举例来说，如果想要以二进制模式读取文件，可以创建一个 FileIO 实例，接着使用 BufferedReader 实例来包裹它。

```
>>> from io import FileIO, BufferedReader
>>> with BufferedReader(FileIO('test.txt', 'r')) as f:
...     print(f.read())
...
b'12345'
>>>
```

FileIO 的模式指定可以是'r'、'w'、'x'、'a'，由于是低级操作，本身就是在处理字节串流，也就没有必须如同 open()函数区分文本模式或二进制模式的需要了。

BufferedReader、BufferedWriter、BufferedRandom 实例的作用就是个包裹器 (Wrapper)，用来包裹 RawIOBase 实例，其中 BufferedRandom 实例继承了 BufferedReader、BufferedWriter，支持 seek()、tell()功能，可进行随机的读取或写入，这要视包裹的 RawIOBase 实例是读取还是写入模式

而定。

如果想要同时进行读取与写入，可以使用 `BufferedRWPair` 同时包裹一个读取与一个写入 `RawIOBase` 实例。

`TextIOWrapper` 也是个包裹器，可以包裹 `BufferedIOBase`，以便将二进制数据按指定的文字编码进行转换，例如：

```
>>> from io import FileIO, BufferedReader, TextIOWrapper
>>> with TextIOWrapper(BufferedReader(FileIO('test_ch.txt', 'r')), 'UTF-8') as f:
...     print(f.read())
...
测试
>>>
```

BytesIO 与 StringIO

如果数据的读取源或写入目标并不是文件，而是内存中某个对象，那么可以使用 `BytesIO` 或 `StringIO`。

`BytesIO` 是 `BufferedIOBase` 的子类，可以直接构建实例，或者指定一个初始的 `bytes` 实例来构建，以便进行数据读取与写入，操作上与文件对象相同。例如：

```
>>> import io
>>> b = io.BytesIO(b'12345')
>>> b.read()
b'12345'
>>> b.seek(2)
2
>>> b.write(b'0')
1
>>> b.seek(0)
0
>>> b.read()
b'12045'
>>> b.close()
>>>
```

通常使用 `BytesIO` 时，最后会使用 `getvalue()` 方法来获取写入的数据。

```
>>> import io
>>> b = io.BytesIO()
>>> b.write(b'1')
1
>>> b.write(b'2')
1
>>> b.write(b'3')
1
>>> b.getvalue()
b'123'
>>> b.close()
>>>
```

类似地，若想读写的是文本数据，可以使用 `StringIO`，它是 `TextIOBase` 的子类。通常使用 `StringIO` 写入数据时会在最后使用 `getvalue()` 来获取数据。

```
>>> import io
>>> s = io.StringIO()
>>> s.write('Line 1\n')
7
>>> s.write('Line 2\n')
```

```

7
>>> s.getvalue()
'Line 1\nLine 2\n'
>>> s.close()
>>>

```

8.3 重点复习

如果想利用 `open()` 函数进行文件读写, 在最基本的需求上, 只需使用它的前两个参数: `file` 与 `mode`。

`open()` 的 `mode` 默认是 `'r'`, 在只指定 `'r'`、`'w'`、`'x'`、`'a'` 的情况下, 就相当于以文本模式打开, 也就等同于 `'rt'`、`'wt'`、`'xt'`、`'at'`, 若要以二进制模式打开, 则要指定 `'rb'`、`'wb'`、`'xb'`、`'ab'`。

如果想以更新模式打开文件, 对于文本模式, 可以使用 `'r+'`、`'w+'`、`'a+'`; 对于二进制模式, 可以使用 `'r+b'`、`'w+b'`、`'a+b'`。

每个被打开的文件, 建议在不使用时, 调用 `open()` 返回的文件对象的 `close()` 方法明确地关闭文件。由于文件对象实现了上下文管理器协议, 因此可使用 `with` 来代替我们进行文件关闭的操作。

`read()` 方法在未指定自变量的情况下会读取文件全部的内容, `read()` 方法在指定整数自变量的情况下会读取指定的字符数或字节数 (视打开模式是文字或者二进制而定)。

对于文本模式来说, 默认读取到 `'\n'`、`'r'` 或 `'r\n'` 都可以被判定为一行, 而 `readline()` 或 `readlines()` 读到的每一行换行字符都一律换为 `'\n'`。对于二进制模式来说, 行的判断标准默认是遇到 `b'\n'` 这个 `bytes` 类型。

文本模式在写入的情况下, 任何 `'\n'` 都会被置换为 `os.linesep` 的值 (Windows 就是 `'\r\n'`)。

Python 的文件读取风格: 读取一个文件最好的方式就是不要去 `read`。

在进行文件读写时, `tell()` 方法可以告知文件指针当前在文件中的位移量, 单位是字节数, 文件开头的位移量是 0, `seek()` 方法可以指定跳到哪个位移量, 可以执行 `flush()` 方法将缓冲内容清出——即清空缓冲区并写入文件。

二进制模式时文件对象拥有一个 `readinto()` 方法接受 `bytearray` 实例, 可以直接将读取到的数据传入。

在 `sys` 模块中有个 `stdin` 代表标准输入, `stdout` 代表标准输出, 它们的操作或行为就像 `open()` 函数打开的文本模式的文件对象。

对于标准输入或输出, 若想要以二进制模式读取或写入, 可以使用 `sys.stdin.buffer` 或 `sys.stdout.buffer`, 它们的操作或行为就像以 `open()` 函数打开的二进制模式的文件对象。

`open()` 函数的 `file` 参数除了接受字符串指定文件的路径之外, 还可以指定文件描述符, 文件描述符是个整数值, 对应到当前程序已打开的文件。

如果能获取一个对应到系统上已打开的文件描述符, 就有机会使用 `open()` 来包裹成为文件对象, 以便使用文件对象的高级操作。

Python 的 I/O 大致上分为三个主要类型: 文本 I/O、二进制 I/O 与原始 I/O, 符合这些分类的具体对象被称为文件对象。

如果数据的读取源或写入目标并不是文件, 而是内存中某个对象, 那么可以使用 `BytesIO` 或 `StringIO`。

课后练习

实践题

1. 请设计一个通用的 `dump()` 函数，可以指定源与目标文件对象或类似文件 (file-like) 的对象，这个函数可以读取源文件对象的内容并写到目标文件对象中。例如，`dump(open('src.jpg', 'rb'), open('dest.jpg', 'wb'))` 最后会创建一个与 `src.jpg` 内容完全相同的 `dest.jpg`，而 `dump(urllib.request.urlopen('http://openhome.cc'), open('index.html', 'wb'))` 将会下载网页。

2. 在例外发生时，可以使用 `traceback.print_exc()` 显示堆栈追踪，如何改写以下程序，使得例外发生时可将堆栈追踪附加至 UTF-8 编码的 `exception.log` 文件。

```
def dump(src_path, dest_path):
    with open(src_path, 'rb') as src, open(dest_path, 'wb') as dest:
        dest.write(src.read())
```

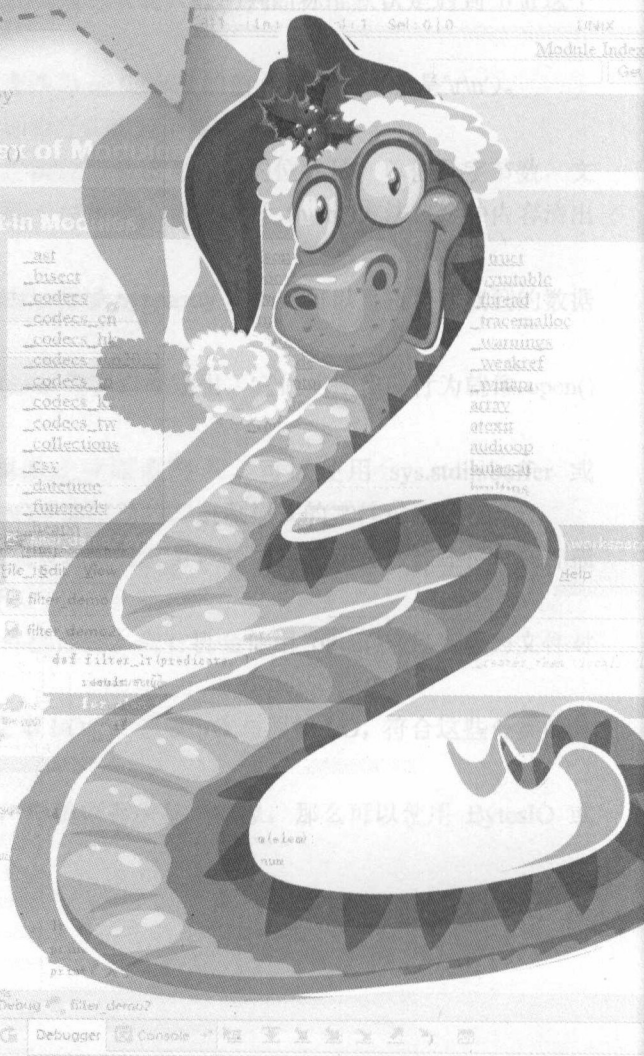
3. 请编写程序，可以指定源文件与编码、将文本文件读入，并以指定的目标文件名转存为 UTF-8 的文本文件。

第 9 章

数据结构

学习目标

- 认识 hashable、iterable、orderable
- 认识群集结构
- 运用 collections 模块
- 运用 collections.abc 模块



9.1 hashable、iterable 与 orderable

在第3章认识 Python 内建类型时，已经看过列表 (list)、集合 (set)、字典 (dict)、元组 (tuple) 等可用来收集集群对象的数据结构。这一章要深入探讨相关的集群，在这之前来看几个在操作集群时可能会遇到的对象协议。

9.1.1 hashable 协议

在 3.1.3 小节中讨论集合 (set) 时说过，集合的内容无序、元素不重复。不过并非任何元素都能放到集合中，例如列表 (list)、字典 (dict) 甚至集合 (set) 本身都不行，试图在集合中放置这些类型的实例就会引发 `TypeError` 错误。

```
>>> {[1, 2, 3]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> {'Justin': 123456}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
>>>
>>> {[1, 2, 3]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
>>>
```

一个对象能被称为 **hashable** (可哈希运算的)，它必须有个 **hash 值** (哈希值)，这个值在整个运行时刻都不会变化，而且必须可以进行相等比较。具体来说，一个对象能被称为 **hashable**，它必须实现 `__hash__()` 与 `__eq__()` 方法。

提示 >>>

可以对一个对象使用 `hash()` 来获取 hash 值，而不是直接调用对象的 `__hash__()` 方法。

Python 的某些链接库在内部需要使用 hash 值，例如集合会对打算加入的对象调用其 `__hash__()` 方法来获取 hash 值，看它是否与当前集合中现有对象的 hash 值都不相同，如果相同就会直接排除而不加入，若 hash 值都不相同，则进一步使用 `__eq__()` 比较相等性，以确定是否要加入到集合之中。

对于 Python 内建类型来说，只要是创建之后状态就无法变动 (**Immutable**) 的类型，它的实例都是 **hashable**，而可变动 (**Mutable**) 类型的实例都是 **unhashable** (不可哈希运算的)。

为什么无法变动的内建类型会默认为 hashable？如果 hash 值是根据对象状态来计算的，对象状态不变基本上计算出来的 hash 值就不变，因此对于无法变动的内建类型就可以根据各自定义好的方式来实现 `__hash__()` 与 `__eq__()` 方法。

一个自定义的类创建的实例默认也是 **hashable** 的，其 `__hash__()` 的实现基本上根据 `id()` 计算而来，而 `__eq__()` 的实现默认使用 `is` 来比较，因此两个分别创建的实例 hash 值必然不相同，而且相等性对比一定不成立。

虽然一个自定义的类创建的实例默认是 hashable，不过放到集合之中或者作为字典 (dict) 的键时，什么样的状态会被认定为重复，还是要自行定义 `__hash__()` 与 `__eq__()`。举例来说：

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def __repr__(self):
...         return 'Point({}, {})'.format(self.x, self.y)
...
>>> p1 = Point(1, 1)
>>> p2 = Point(2, 2)
>>> p3 = Point(1, 1)
>>> ps = {p1, p2, p3}
>>> ps
{Point(1, 1), Point(2, 2), Point(1, 1)}
>>>
```

从这里的例子中可以看到，虽然 p1 与 p3 代表的是相同坐标，然而在集合中两个都收纳了，这是因为 p1 与 p3 使用默认的 `__hash__()` 获取的 hash 值不同，而默认的 `__eq__()` 比较也是不相等的结果。

如果想让集合能剔除代表相同坐标的 Point 对象，必须自行实现 `__eq__()` 与 `__hash__()` 方法。例如：



object_protocols point_demo.py

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, that):
        if hasattr(that, 'x') and hasattr(that, 'y'):
            return self.x == that.x and self.y == that.y
        return False

    def __hash__(self):
        return 41 * (41 + self.x) + self.y

    def __str__(self):
        return self.__repr__()

    def __repr__(self):
        return 'Point({}, {})'.format(self.x, self.y)

p1 = Point(1, 1)
p2 = Point(2, 2)
p3 = Point(1, 1)
ps = {p1, p2, p3}
print(ps) # 显示 {Point(1, 1), Point(2, 2)}
```

在上面的范例中，除了定义 `__hash__()` 之外，还定义了 Point 的相等性，必须是 x 与 y 都相同才行，因此在最后的结果显示中可以看到集合中并不会包含相同的坐标。

提示 >>>

当集合判断新加入的对象与已经包含的某对象的 hash 值相同，而且相等性比较也成立时，就会丢弃已包含的对象，并将新的对象加入。

现在有个有趣的问题是，在上面的范例中，如果在 `print(ps)` 之后，又加上 `p2.x = 1`、`p2.y = 1` 这两行程序代码，并且再度 `print(ps)` 会怎么样呢？我们会看到 `{Point(1, 1), Point(1, 1)}` 的显示结果，为什么？因为集合判定是否重复是在对象加入时，当对象已经在集合中了，我们又通过其他方式改变了对象状态，就会造成这种尴尬的情况，这也就是为什么 **hashable** 对象建议状态是不可变动的。在这里的例子中，必要时可为 `Point` 加上一些存取限制的定义。

两个对象若是相等性比较成立，那么也必须有相同的 **hash** 值，然而 **hash** 值相同，两个对象的相等性比较不一定是成立的。

9.1.2 iterable 协议

Python 提供了 `for in` 语句，不少内建类型或生成器（像列表（list）、字符串（str）、元组（tuple）、字典（dict）甚至是文件对象）都可以使用 `for in` 来进行迭代。实际上，只要对象具有 `__iter__()` 方法，可返回一个迭代器（Iterator），具有 `__iter__()` 方法的对象，就是一个 **iterable**（可迭代的）对象。

可以使用 `iter()` 方法从一个对象获取迭代器，而不用调用对象的 `__iter__()` 方法，返回的迭代器具有 `__next__()` 方法，可以逐一迭代出对象中的信息，若无法进一步迭代，会引发 `StopIteration` 例外。迭代器也会具有 `__iter__()` 方法，返回迭代器自身，因此每个迭代器本身也是个 **iterable** 对象。

在 4.2.6 小节中讨论过生成器，生成器也是一种迭代器，对于大部分的迭代需求，使用 `yield` 语句创建生成器会比较简单而直接。举个例子来说，想要创建一个迭代器，对指定的序列不断地重复进行迭代，例如对 `cycle('abcd')` 进行迭代，结果是不断地 'a'、'b'、'c'、'd'、'a'、'b'、'c'、'd'…… 循环下去，这个需求可以如下实现：

```
>>> def cycle(elems):
...     while True:
...         for elem in elems:
...             yield elem
...
>>> abcd_gen = cycle('abcd')
>>> next(abcd_gen)
'a'
>>> next(abcd_gen)
'b'
>>> next(abcd_gen)
'c'
>>> next(abcd_gen)
'd'
>>> next(abcd_gen)
'a'
>>> next(abcd_gen)
'b'
>>>
```

实现 `__iter__()`

对于状态比较复杂的对象，在生成器不适合时就要自己实现 `__iter__()` 等方法来创建迭代器。为了示范如何以 `__iter__()` 实现 **iterable** 对象，下面的范例故意不使用 `yield` 来实现。

object_protocols tools.py

```
def cycle(elems):
```

```

while True:
    for elem in elems:
        yield elem

class Repeat:
    def __init__(self, elem, n):
        self.elem = elem
        self.n = n

    def __iter__(self):          ← ①实现__iter__()方法
        elem = self.elem
        n = self.n

    class _Iter:                ← ②定义迭代器
        def __init__(self):
            self.count = 0

        def __next__(self):      ← ③定义__next__()方法
            if self.count < n:
                self.count += 1
                return elem
            else:
                raise StopIteration ← ④引发 StopIteration 例外停止迭代

        def __iter__(self):      ← ⑤迭代器的__iter__()返回自身
            return self

    return _Iter()              ← ⑥返回迭代器实例

for elem in Repeat('A', 5):
    print(elem, end = ' ')

```

在这个范例中，Repeat 类定义了__iter__()方法①，它必须返回迭代器实例⑥，迭代器类直接定义在 Repeat 类中②是为了便于存取 Repeat 的成员；迭代器定义了__next__()方法③，不断地重复返回指定的元素，如果返回的元素已达指定个数，就引发 StopIteration 例外停止迭代④，而迭代器的__iter__()返回自身⑤。这个范例的执行结果最后会显示 AAAAA 的字样。

当然，这只是个示范，同样的需求也可以使用生成器来实现。例如：

```

def repeat(elem, n):
    count = 0
    while count < n:
        count += 1
        yield elem

for elem in repeat('A', 5):
    print(elem, end = ' ')

```

提示 >>>

如果需要的重复次数不多也可以使用[elem] * n 的方式，例如['A'] * 10 会创建['A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A']，不过这会返回列表而不是生成器，因此若需要的重复次数多时，会比较耗费内存空间。

实际上很少有机会直接调用__iter__()，或者使用 iter() 获取生成器，因为 Python 标准链接库在许多情况下都接受 iterable 对象，在内部自动帮你调用__iter__()。举例来说，若 lt 是 [1, 2, 3, 4, 5]，

那么 `set(lt)` 会创建 `{1, 2, 3, 4, 5}`, `tuple(lt)` 会创建 `(1, 2, 3, 4, 5)`。

使用 `itertools` 模块

在 Python 标准链接库中提供了 `itertools` 模块, 当中有许多函数可协助创建迭代器或生成器。例如刚才实现的 `cycle()`、`repeat()` 函数, 在 `itertools` 模块中就提供了。

```
>>> import itertools
>>> cycle_gen = itertools.cycle('abcd')
>>> next(cycle_gen)
'a'
>>> next(cycle_gen)
'b'
>>> next(cycle_gen)
'c'
>>> next(cycle_gen)
'd'
>>> next(cycle_gen)
'a'
>>> rept_gen = itertools.repeat('A', 10)
>>> list(rept_gen)
['A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A']
>>>
```

在 `itertools` 模块中, `cycle()`、`repeat()` 还有一个 `count()` 函数都是无限迭代器 (`repeat()` 的第二个自变量可以省略, 此时就会创建无限生成器)。例如 `count()` 可以指定起始值与步进值, 无限地迭代出下一个数字, 像 `count(5)` 可以迭代出 5、6、7、8、9……而 `count(5, 2)` 可以迭代出 5、7、9、11、13……

对于迭代过程的一些常见操作, `itertools` 模块也提供了相关的函数, 例如 `accumulate()` 可在迭代的过程中进行累加或指定的运算。

```
>>> import itertools
>>> list(itertools.accumulate([1, 2, 3, 4, 5]))
[1, 3, 6, 10, 15]
>>> list(itertools.accumulate([1, 2, 3, 4, 5], int.__mul__))
[1, 2, 6, 24, 120]
>>>
```

`chain()` 或 `chain.from_iterable()` 可将指定的序列摊平逐一迭代。例如:

```
>>> list(itertools.chain('ABC', [1, 2, 3]))
['A', 'B', 'C', 1, 2, 3]
>>> list(itertools.chain.from_iterable(['ABC', [1, 2, 3]]))
['A', 'B', 'C', 1, 2, 3]
>>> list(itertools.chain.from_iterable([[9, 8, 6], [1, 2, 3]]))
[9, 8, 6, 1, 2, 3]
>>>
```

`dropwhile()` 会在指定的函数返回 `True` 的情况下持续地丢弃元素, 直到有个元素让函数返回 `False` 为止, `takewhile()` 则和它相反, 持续地保留元素, 直到有个元素让函数返回 `False` 为止; `filterfalse()` 是 `filter()` 函数的相反函数, `filterfalse()` 会将令指定函数返回 `False` 的元素留下来。

```
>>> list(itertools.dropwhile(lambda x: x < 5, [1, 4, 6, 4, 1]))
[6, 4, 1]
>>> list(itertools.takewhile(lambda x: x < 5, [1, 4, 6, 4, 1]))
[1, 4]
>>> list(itertools.filterfalse(lambda x: x % 2, [1, 2, 3, 4]))
[2, 4]
>>>
```


有时候可能需要按某个键来进行分类，例如，将['Justin', 'Monica', 'Irene', 'Pika', 'caterpillar']字符串按长度分类，就这个需求而言，虽然可以自己实现，例如：

```
names = ['Justin', 'Monica', 'Irene', 'Pika', 'caterpillar']

grouped_by_len = {}
for name in names:
    key = len(name)
    if key not in grouped_by_len:
        grouped_by_len[key] = []
    grouped_by_len[key].append(name)

for length in grouped_by_len:
    print(length, grouped_by_len[length])
```

不过，使用 `itertools` 的 `groupby()` 函数可以省事许多。

```
>>> names = ['Justin', 'Monica', 'Irene', 'Pika', 'caterpillar']
>>> grouped_by_name = itertools.groupby(names, lambda name: len(name))
>>> for length, group in grouped_by_name:
...     print(length, list(group))
...
6 ['Justin', 'Monica']
5 ['Irene']
4 ['Pika']
11 ['caterpillar']
>>>
```

`groupby()` 的返回值是个 `itertools.groupby` 对象，它是 `iterable` 对象，从它身上获取的迭代器在每次迭代时会返回一个元组（`tuple`），元组中第一个值就是指定的分类值，第二个值是个 `itertools._grouper`，也是个 `iterable` 对象，当中包含了所有同一分类值的对象。

这里先介绍 `itertools` 模块中几个常用的函数，更多的函数说明可以参考 `itertools` 模块¹的帮助文档。

提示 >>>

记得在 8.1.1 小节中曾经介绍过 Python 的文件读取风格：读取一个文件最好的方式就是不要去 `read`。由于 Python 标准链接库在许多情况下都接受 `iterable` 对象，而使用 `open()` 打开的文件对象是 `iterable` 对象，直接将 `open()` 打开的文件传给这类链接库就非常方便了。例如，若想读取文本文件的内容，并将每一行安插到集合中，一个简洁的方式就是：

```
unrepeated_lines = None
with open('filename', 'r') as f:
    unrepeated_line = set(f)
```

9.1.3 orderable 协议

如果打算对一个列表进行排序，可以直接调用它的 `sort()` 方法，这会在现有的列表上进行排序，例如：

¹ `itertools` 模块：docs.python.org/3.5/library/itertools.html

```
>>> lt = [3, 5, 1, 2, 8]
>>> lt.sort()
>>> lt
[1, 2, 3, 5, 8]
>>> lt.sort(reverse = True)
>>> lt
[8, 5, 3, 2, 1]
>>>
```

除了可以使用 `reverse` 参数指定反序之外,也可以使用 `key` 参数,指定使用哪个值进行排序。例如下面分别针对姓名、代表字母或年龄进行排序。

```
>>> customers = [('Justin', 'A', 40), ('Irene', 'C', 8), ('Monica', 'B', 37)]
>>> customers.sort(key = lambda cust: cust[0])
>>> customers
[('Irene', 'C', 8), ('Justin', 'A', 40), ('Monica', 'B', 37)]
>>> customers.sort(key = lambda cust: cust[1])
>>> customers
[('Justin', 'A', 40), ('Monica', 'B', 37), ('Irene', 'C', 8)]
>>> customers.sort(key = lambda cust: cust[2])
>>> customers
[('Irene', 'C', 8), ('Monica', 'B', 37), ('Justin', 'A', 40)]
>>>
```

列表才有 `sort()` 方法,对于其他 `iterable` 对象,若想进行排序,可以使用 `sorted()` 函数,可指定的参数同样也有 `reverse` 与 `key` 参数,此函数不会变动原有的函数,排序的结果会以新的列表返回。例如:

```
>>> customers = [('Justin', 'A', 40), ('Irene', 'C', 8), ('Monica', 'B', 37)]
>>> sorted(customers, key = lambda cust: cust[0])
[('Irene', 'C', 8), ('Justin', 'A', 40), ('Monica', 'B', 37)]
>>> sorted(customers, key = lambda cust: cust[2])
[('Irene', 'C', 8), ('Monica', 'B', 37), ('Justin', 'A', 40)]
>>> sorted(customers, key = lambda cust: cust[2], reverse = True)
[('Justin', 'A', 40), ('Monica', 'B', 37), ('Irene', 'C', 8)]
>>>
```

无论是使用列表的 `sort()` 方法还是 `sorted()` 函数,有一个问题就是如果是自定义的类实例,它们怎么会知道该怎么排序呢?确实是不知道的!

```
>>> class Customer:
...     def __init__(self, name, symbol, age):
...         self.name = name
...         self.symbol = symbol
...         self.age = age
...
>>> customers = [
...     Customer('Justin', 'A', 40),
...     Customer('Irene', 'C', 8),
...     Customer('Monica', 'B', 37)
... ]
>>> sorted(customers)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Customer() < Customer()
>>>
```

如果没有指定 `key` 参数,在上面的范例中 `sorted()` 根本不知道排序的根据是什么,因此引发了 `TypeError` 错误,告知 `Customer` 并不是 `orderable`(可排序的)类型。如果希望使用自定义类型的 `sorted()`

或者使用列表的 `sort()` 时有默认的排序定义，那么必须实现 `__lt__()` 方法。例如，若想让名称作为默认排序的依据，可以如下实现：



object_protocols/orderable_types.py

```
class Customer:
    def __init__(self, name, symbol, age):
        self.name = name
        self.symbol = symbol
        self.age = age

    def __lt__(self, other):
        return self.name < other.name

    def __str__(self):
        return "Customer('{name}', '{symbol}', {age})".format(**vars(self))

    def __repr__(self):
        return self.__str__()

customers = [
    Customer('Justin', 'A', 40),
    Customer('Irene', 'C', 8),
    Customer('Monica', 'B', 37)
]

print(sorted(customers))
```

当然，对于一个复合类型的实例来说，排序时可能有多种考虑，因此在使用列表的 `sort()` 方法或者 `sorted()` 函数时指定 `key` 参数还是比较方便的。在指定 `key` 参数时，虽然可以自行定义 `lambda` 来指定排序的依据，不过也可以指定 `operator` 模块的 `itemgetter`、`attrgetter`，前者可以针对具有索引的结构，后者可以针对对象的属性。下面是使用 `itemgetter` 的示范：

```
>>> from operator import itemgetter
>>> customers = [('Justin', 'A', 40), ('Irene', 'C', 8), ('Monica', 'B', 37)]
>>> sorted(customers, key = itemgetter(0))
[('Irene', 'C', 8), ('Justin', 'A', 40), ('Monica', 'B', 37)]
>>> sorted(customers, key = itemgetter(1))
[('Justin', 'A', 40), ('Monica', 'B', 37), ('Irene', 'C', 8)]
>>> sorted(customers, key = itemgetter(2))
[('Irene', 'C', 8), ('Monica', 'B', 37), ('Justin', 'A', 40)]
>>>
```

下面是使用 `attrgetter` 的示范：

```
>>> from operator import attrgetter
>>> class Customer:
...     def __init__(self, name, symbol, age):
...         self.name = name
...         self.symbol = symbol
...         self.age = age
...     def __repr__(self):
...         return "Customer('{name}', '{symbol}', {age})".format(**vars(self))
...
>>> customers = [
...     Customer('Justin', 'A', 40),
...     Customer('Irene', 'C', 8),
...     Customer('Monica', 'B', 37)
... ]
>>> sorted(customers, key = attrgetter('name'))
```



```
[Customer('Irene', 'C', 8), Customer('Justin', 'A', 40), Customer('Monica', 'B', 37)]
>>> sorted(customers, key = attrgetter('age'))
[Customer('Irene', 'C', 8), Customer('Monica', 'B', 37), Customer('Justin', 'A', 40)]
>>> sorted(customers, key = attrgetter('symbol'))
[Customer('Justin', 'A', 40), Customer('Monica', 'B', 37), Customer('Irene', 'C', 8)]
>>>
```

9.2 高级群集处理

在第3章介绍字符串(str)、列表(list)、集合(set)、字典(dict)、元组(tuple)等类型时应该会注意到不同类型之间都有一些相同的操作或行为。实际上,Python 是个动态类型的程序设计语言,许多时候并不是以类型来分类,而是以操作(或行为)来分类,从这个角度来认识群集架构就可以活用群集,不至于落入死记硬背 API 的窘境。

9.2.1 认识群集结构

想要进一步认识群集,先要知道在 Python 中大致将群集分为三种类型:序列类型(Sequences type)、集合类型(Set type)与映射类型(Mapping type)。

序列类型

目前我们看过的序列类型有列表(list)、元组(tuple)、范围(range),以及代表文字数据的字符串(str)和代表二进制数据的 bytes、bytearray 等。可以看出,序列类型都是有序、具备索引的数据结构,序列类型都是 iterable 对象,都具有表 9.1 所示的操作(或行为)。

表 9.1 序列类型的共同操作

操作	结果
<code>x in s</code>	s 中是否包含 x 元素
<code>x not in s</code>	s 中是否未包含 x 元素
<code>s + t</code>	串接 s 与 t
<code>s * n</code>	将 s 的元素重复 n 次
<code>s[i]</code>	获取索引 i 处的元素,第一个索引是 0
<code>s[i:j]</code>	分割出从 i 到 j 的元素
<code>s[i:j:k]</code>	分割出从 i 到 j 的元素,每次间隔 k
<code>len(s)</code>	获取 s 的长度
<code>min(s)</code>	获取 s 中的最小值
<code>max(s)</code>	获取 s 中的最大值
<code>s.index(x, [i, j])</code>	获取第一个 x 的索引位置(可指定从 i 开始,至 j 之前)
<code>s.count(x)</code>	获取 x 的出现次数

元组、字符串与 bytes 是不可变动的序列类型,具有默认的 hash()实现,其他可变动的序列类型就没有默认的 hash()实现,因此元组、字符串、bytes 可以作为集合的元素或字典的键,然而可变动的列表就不行。

可变动的序列结构还会有表 9.2 所示的操作(或行为)。

表 9.2 可变动序列类型的共同操作

操作	结果
<code>s[i] = x</code>	将 <code>s</code> 的索引 <code>i</code> 处赋值为 <code>x</code>
<code>s[i:j] = t</code>	将 <code>s</code> 的 <code>i</code> 至 <code>j</code> 使用 <code>iterable</code> 的 <code>t</code> 取代
<code>del s[i:j]</code>	相当于 <code>s[i:j] = []</code>
<code>s[i:j:k] = t</code>	将符合 <code>s[i:j:k]</code> 的元素用 <code>t</code> 取代
<code>del s[i:j:k]</code>	将 <code>s[i:j:k]</code> 的元素删除
<code>s.append(x)</code>	将 <code>x</code> 附加至 <code>s</code> 尾端
<code>s.clear()</code>	清空 <code>s</code> 中的全部元素
<code>s.copy()</code>	浅层复制 <code>s</code> (相当于 <code>s[:]</code>)
<code>s.extend(t)</code>	相当于 <code>s += t</code>
<code>s.insert(i, x)</code>	将 <code>x</code> 插入到索引 <code>i</code> 之前
<code>s.pop([i])</code>	获取首个 (或索引 <code>i</code>) 元素并将之从 <code>s</code> 中删除
<code>s.remove(x)</code>	将 <code>x</code> 从 <code>s</code> 中删除
<code>s.reverse()</code>	反转 <code>s</code> 中元素的顺序

提示 >>>

如果需要一个仅含同质 (Homogeneous) 元素的序列结构, 可以使用 `array` 模块¹ 的 `array` 类。例如, 创建一个只允许整数的 `array` 实例, 如果指定了非整数就会引发 `TypeError` 错误。

```
>>> from array import array
>>> ints = array('i', [10, 20, 30])
>>> ints.append(40)
>>> ints.append('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: an integer is required (got type str)
>>>
```

● 集合类型

集合类型是无序的, 而且元素必须都是 `hashable` 对象而且不会重复, 它们是 `iterable` 对象, 可以使用 `x in set`、`x not in set`、`len(set)` 以及交集 (`Intersection`)、并集 (`Union`)、差集 (`Difference`) 与对称差集 (`Symmetric difference`) 等操作。

集合类型的内建类型是集合 (`set`), 除了使用 3.2.5 小节中介绍的 `&`、`|`、`-` 与 `^` 来进行交集、并集、差集与对称差集运算, 还可以使用 `intersection()`、`union()`、`difference()`、`symmetric_difference()` 方法来进行运算。

集合本身是可变动的, 如果想要不可变动的集合类型, 可以使用 `frozenset()` 来创建, 创建的实例本身实现了 `__hash__()` 方法, 即为 `hashable` 对象。

对于 `set` 与 `frozenset()` 创建的集合, 它们拥有共同的操作或行为, 可以参考官方文件 “Set Types”²

¹ `array` 模块: docs.python.org/3/library/array.html

² Set Types: docs.python.org/3.5/library/stdtypes.html#set-types-set-frozenset

——set, frozenset”的说明，其中也有可变动的集合才能操作的相关方法的说明。

映射类型

映射类型可以将 hashable 对象映射至一个任意值，Python 中的内建类型就是字典（dict），操作方式可参考 3.1.3 小节有关字典的介绍，或者参考官方文件“Mapping Types¹——dict”的说明。

9.2.2 使用 collection 模块

除了内建类型之外，Python 标准链接库中还包含了 collections 模块，其中包含了一些群集相关的函数与方法，可用来满足一些群集处理的高级需求。

deque 类

如果想实现先进后出的堆栈（Stack）结构，在 Python 中可以使用列表，运用其 append() 与 pop() 方法。例如：

```
>>> stack = [1, 2, 3]
>>> stack.append(4)
>>> stack.append(5)
>>> stack
[1, 2, 3, 4, 5]
>>> stack.pop()
5
>>> stack
[1, 2, 3, 4]
>>> stack.pop()
4
>>> stack
[1, 2, 3]
>>>
```

如果想要实现先进先出的队列（Queue），在 Python 中也可以使用列表，运用 append() 与 pop(0) 方法，或者想实现双向队列（Double-ended queue），可在队列前端或末端插入或获取元素，列表也使用提供的 insert(0, elem) 方法。

不过，对于队列或双向队列来说，使用列表的效率并不好，因为列表本身的实现对于固定长度的存取会比较快，若使用 pop(0) 或 insert(0, elem) 方法，为了维持索引顺序，必须进行 $O(n)$ 数量级的元素搬动，若列表长度很长，或者必须频繁执行 pop(0)、insert(0, elem) 操作，不建议使用列表。

对于队列或双向队列的需求，建议使用 collections 模块中提供的 deque 类，在 deque 实例的两端执行插入、删除的操作几乎是接近 $O(1)$ 数量级的性能。除了与列表相同的 append()、pop()、insert() 等方法之外，deque 还提供了 appendleft()、popleft() 等方法。例如：

```
>>> from collections import deque
>>> deque = deque([1, 2, 3])
>>> deque.appendleft(0)
>>> deque.appendleft(-1)
>>> deque
deque([-1, 0, 1, 2, 3])
```

¹ Mapping Types: docs.python.org/3.5/library/stdtypes.html#mapping-types-dict


```
>>> deque.pop()
3
>>> deque.popleft()
-1
>>> deque
deque([0, 1, 2])
>>>
```

deque 甚至还有个 `rotate()` 方法, 可以实现环形队列, `rotate()` 可以指定一次转几个元素, 例如一次转一个元素。

```
>>> deque
deque([0, 1, 2])
>>> deque.rotate(1)
>>> deque
deque([2, 0, 1])
>>> deque.rotate(1)
>>> deque
deque([1, 2, 0])
>>>
```

🔗 namedtuple() 函数

在 3.1.3 小节介绍元组 (tuple) 时曾经说明过, 有时想要返回一组相关的值, 又不想特地自定义一个类型, 就可以使用元组, 这有一些好处, 元组的状态不可变动, 比较省内存, 而且元组是 hashable (可哈希运算的), 可以作为集合的元素或字典 (dict) 的键。

不过, 元组的元素没有名称, 只能依靠索引来获取各个元素并不方便, 如果想要有个简单类, 以便创建的实例能拥有字段名, 实际上不用自行定义, 可以使用 `collections` 模块的 `namedtuple()` 函数。例如:

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p1 = Point(10, 20)
>>> p1.x
10
>>> p1.y
20
>>> p2 = Point(11, y = 22)
>>> p2
Point(x=11, y=22)
>>> x, y = p1
>>> x
10
>>> y
20
>>> p2[0]
11
>>> p2[1]
22
>>> p1.x = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> hash(p2)
112275502
>>>
```

`namedtuple()` 的第一个参数是想要创建的类型名称, 第二个参数是字段名, 它会返回元组的子

类，如上面的例子所示，可以用它来创建实例，并具有字段名，同时也保有元组的特性，比如状态无法变动，且有默认的 `__hash__()` 实现。

提示 >>>

Python 的创建者 Guido van Rossum 曾经写道：“避免过度设计数据结构。元组比对象好（试试 `namedtuple`）。简单的字段会比 Getter/Setter 函数好。”（<http://goo.gl/NMGKOj>）

除了继承元组可用的方法之外，`namedtuple()` 返回的类创建的实例也额外定义了一些方法可以使用，为了避免与用户指定的字段名冲突，这些方法的名称都是以下划线作为开头的。

例如刚才的 `Point` 类，如果来源是个 iterable 对象，除了 `Point(*iterable)` 这样的方式之外，还可以使用 `Point._make(iterable)` 的方式来创建 `Point` 实例。

```
>>> lt = [10, 20]
>>> Point(*lt)
Point(x=10, y=20)
>>> Point._make(lt)
Point(x=10, y=20)
>>>
```

可以使用 `_asdict()` 方法返字段名与值，使用 `_replace()` 指定字段与值会创建新的实例，当中包含已取代的字段值。

```
>>> p1._asdict()
OrderedDict([('x', 10), ('y', 20)])
>>> p1._replace(x = 20)
Point(x=20, y=20)
>>>
```

通过 `namedtuple()` 返回的类有个 `_fields` 可获取全部字段名，如果想简单地定义 `namedtuple` 的继承，可以如下编写：

```
>>> Point._fields
('x', 'y')
>>> Point3D = namedtuple('Point3D', Point._fields + tuple('z'))
>>> Point3D(10, 20, 30)
Point3D(x=10, y=20, z=30)
>>>
```

如果想要定义 Docstrings，可以直接定义 `namedtuple()` 返回类或者是其字段的 `__doc__`，例如：

```
>>> Point.__doc__ = 'Cartesian coordinate system (x, y)'
>>> Point.x.__doc__ = 'Cartesian coordinate system x'
>>> Point.y.__doc__ = 'Cartesian coordinate system y'
>>>
```

由于 `namedtuple()` 返回的实际上就是个类，因此也可以直接使用继承语句。若想定义一个类，状态不可变动，又可以拥有一些自定义的方法，可以如下编写：

```
>>> from math import sqrt
>>> from collections import namedtuple
>>>
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     def len_from(self, other):
...         return sqrt(pow(self.x - other.x, 2) + pow(self.y - other.y, 2))
...
>>> p1 = Point(5, 10)
```

```
>>> p2 = Point(8, 15)
>>> p1.len_from(p2)
5.830951894845301
>>>
```

OrderedDict 类

在使用内建类型字典 (dict) 时, 必须遍历键时就会发现无法测试它的顺序, 如果想以一定的顺序来遍历字典中的键值, 那么获取字典全部键的排序后再用来取值是可行的。

```
>>> custs = [
...     ('A', 'Justin Lin'),
...     ('B', 'Monica Huang'),
...     ('C', 'Irene Lin')
... ]
>>>
>>> cust_dict = dict(custs)
>>> cust_dict
{'C': 'Irene Lin', 'B': 'Monica Huang', 'A': 'Justin Lin'}
>>> for key in sorted(cust_dict.keys()):
...     print(key, ': ', cust_dict[key])
...
A : Justin Lin
B : Monica Huang
C : Irene Lin
>>>
```

不过, 既然一开始的 `custs` 数据就有这样的顺序, 事后又得对字典的键进行排序显得有点麻烦, 如果想要在创建字典时保有最初键值加入的顺序, 可以使用 `collections` 模块的 `OrderedDict`。

```
>>> custs = [
...     ('A', 'Justin Lin'),
...     ('B', 'Monica Huang'),
...     ('C', 'Irene Lin')
... ]
>>>
>>> cust_dict = OrderedDict(custs)
>>> cust_dict
OrderedDict([('A', 'Justin Lin'), ('B', 'Monica Huang'), ('C', 'Irene Lin')])
>>> for key in cust_dict:
...     print(key, ': ', cust_dict[key])
...
A : Justin Lin
B : Monica Huang
C : Irene Lin
>>>
```

`OrderedDict` 搭配 9.1.3 小节的内容就可以解决各种按键排序或按值排序的常见需求。

```
>>> from operator import itemgetter
>>> origin_dict = {'A': 85, 'B': 90, 'C': 70}
>>> origin_dict
{'C': 70, 'B': 90, 'A': 85}
>>> OrderedDict(sorted(origin_dict.items(), key = itemgetter(0))) # 按键排序
OrderedDict([('A', 85), ('B', 90), ('C', 70)])
>>> OrderedDict(sorted(origin_dict.items(), key = itemgetter(1))) # 按值排序
OrderedDict([('C', 70), ('A', 85), ('B', 90)])
>>>
```

defaultdict 类

回顾一下, 在 9.1.2 小节介绍 `itertools` 模块中的 `groupby()` 函数之前曾经使用过的范例:


```
names = ['Justin', 'Monica', 'Irene', 'Pika', 'caterpillar']

grouped_by_len = {}
for name in names:
    key = len(name)
    if key not in grouped_by_len:
        grouped_by_len[key] = []
    grouped_by_len[key].append(name)

for length in grouped_by_len:
    print(length, grouped_by_len[length])
```

如果只是讨论这段程序代码，那么粗体字的部分可以改写为：

```
names = ['Justin', 'Monica', 'Irene', 'Pika', 'caterpillar']

grouped_by_len = {}
for name in names:
    key = len(name)
    group = grouped_by_len.get(key, [])
    group.append(name)
    grouped_by_len[key] = group

for length in grouped_by_len:
    print(length, grouped_by_len[length])
```

这里使用了字典实例的 `get()` 方法，在指定的键不存在时返回第二个参数指定的值。实际上对这个需求来说，只要指定的键不存在，一律返回新建的列表。对于这种场合，另一个方式是使用 `collections` 的 `defaultdict` 类。例如：

collection_advanced group.py

```
from collections import defaultdict

names = ['Justin', 'Monica', 'Irene', 'Pika', 'caterpillar']

grouped_by_len = defaultdict(list)

for name in names:
    key = len(name)
    grouped_by_len[key].append(name)

for length in grouped_by_len:
    print(length, grouped_by_len[length])
```

`defaultdict` 接受一个函数，它创建的实例在指定的键不存在时，就会使用指定的函数来产生，并直接设置为键的对应值。在上面的范例中，指定了列表在键不存在时产生一个空的列表并设置为键对应的值。

也可以使用 `defaultdict` 来设计一个计数器，例如计算文字中每个字母出现的次数。

collection_advanced counter.py

```
from collections import defaultdict
from operator import itemgetter

def count(text):
    counter = defaultdict(int)
    for c in text:
```

```

        counter[c] += 1
    return counter.items()

text = 'Your right brain has nothing left.'
for c, n in sorted(count(text), key = itemgetter(0)):
    print(c, ': ', n)

```

如果指定的字母不存在，就会使用 `int` 产生默认的整数值 0，并设为键对应的值，之后就直接加 1 并再度设回键对应的值。执行结果如下：

```

: 5
. : 1
Y : 1
a : 2
b : 1
e : 1
f : 1
g : 2
...

```

Counter 类

实际上 `collections` 模块中就有个 `Counter` 类，可以满足刚才的计数需求。例如：

```

>>> from collections import Counter
>>> c = Counter('Your right brain has nothing left.')
>>> c
Counter({' ': 5, 'h': 3, 'i': 3, 'r': 3, 't': 3, 'n': 3, 'a': 2, 'o': 2, 'g': 2, 'l': 1, 'f': 1, 'b': 1, 'Y': 1, 'e': 1, 's': 1, 'u': 1, '.': 1})
>>> list(c.elements())
['h', 'h', 'h', 'i', 'i', 'i', 'l', 'f', 'r', 'r', 'r', 'b', 'Y', 'e', 'a', 'a', 's', 'u', '.', 't', 't', 't', 'o', 'o', 'n', 'n', 'n', ' ', ' ', ' ', ' ', ' ', ' ', 'g', 'g']
>>>

```

反过来运用也可以指定一个字典 (dict) 给 `Counter`，它会按字典中值的指定创建对应数量的键。例如：

```

>>> c = Counter({'Justin' : 4, 'Monica' : 3, 'Irene' : 2})
>>> list(c.elements())
['Monica', 'Monica', 'Monica', 'Justin', 'Justin', 'Justin', 'Justin', 'Irene', 'Irene']
>>> c['Justin'] = 5
>>> list(c.elements())
['Monica', 'Monica', 'Monica', 'Justin', 'Justin', 'Justin', 'Justin', 'Justin', 'Justin', 'Irene', 'Irene']
>>> c['caterpillar'] = 2
>>> list(c.elements())
['Monica', 'Monica', 'Monica', 'Justin', 'Justin', 'Justin', 'Justin', 'Justin', 'Irene', 'Irene', 'caterpillar', 'caterpillar']
>>>

```

由于 `Counter` 本身是字典 (dict) 的子类，因此想要新增或删除键值，方式与字典都是相同的。

ChainMap 类

如果有多个字典对象想要将它们合并在一起，可以使用字典的 `update()` 方法。例如：

```

>>> custs1 = {'A' : 'Justin', 'B' : 'Monica'}
>>> custs2 = {'C' : 'Irene', 'D' : 'caterpillar'}
>>> custs = {}
>>> custs.update(custs1)
>>> custs.update(custs2)

```

```
>>> custs
{'D': 'caterpillar', 'B': 'Monica', 'C': 'Irene', 'A': 'Justin'}
>>>
```

也可以使用 collections 的 ChainMap 来达到相同的目的, ChainMap 的实例操作有如字典, 可以将多个字典视为一个来进行操作。例如:

```
>>> from collections import ChainMap
>>> custs1 = {'A': 'Justin', 'B': 'Monica'}
>>> custs2 = {'C': 'Irene', 'D': 'caterpillar'}
>>> custs = ChainMap(custs1, custs2)
>>> custs
ChainMap({'B': 'Monica', 'A': 'Justin'}, {'D': 'caterpillar', 'C': 'Irene'})
>>> custs['B']
'Monica'
>>> custs['D']
'caterpillar'
>>>
```

从上面的操作中也可以看到, 实际上 ChainMap 的底层使用了一个列表来维护最初指定的全部字典, 因此它会比单纯使用字典的 update() 来合并多个字典更有效率。

如果通过 ChainMap 指定更新某对键值, 会在底层中第一个找到键的字典中更新对应的值, 若底层全部的字典都找不到对应的键时, 就会直接在第一个字典新增键值。例如:

```
>>> custs1 = {'A': 'Justin', 'B': 'Monica'}
>>> custs2 = {'B': 'Irene', 'C': 'caterpillar'}
>>> custs = ChainMap(custs1, custs2)
>>> custs
ChainMap({'B': 'Monica', 'A': 'Justin'}, {'B': 'Irene', 'C': 'caterpillar'})
>>> custs['B'] = 'Pika'
>>> custs
ChainMap({'B': 'Pika', 'A': 'Justin'}, {'B': 'Irene', 'C': 'caterpillar'})
>>> custs['D'] = 'Bush'
>>> custs
ChainMap({'D': 'Bush', 'B': 'Pika', 'A': 'Justin'}, {'B': 'Irene', 'C': 'caterpillar'})
>>>
```

ChainMap 底层维护的列表可以通过 maps 属性来获取, 这是一个列表, 因此只要使用索引就可以获取对应的字典。

```
>>> custs.maps
[{'D': 'Bush', 'B': 'Pika', 'A': 'Justin'}, {'B': 'Irene', 'C': 'caterpillar'}]
>>> custs.maps[0]
{'D': 'Bush', 'B': 'Pika', 'A': 'Justin'}
>>>
```

如果想要在现有的 ChainMap 中新增字典, 方式是在 maps 属性上使用 append() 方法。ChainMap 的 new_child() 方法可以指定字典, 这会创建一个新的 ChainMap, 当中含有源 ChainMap 中的字典并包含指定的字典。如果想创建一个新的 ChainMap, 当中不包含源 ChainMap 的第一个字典, 可以使用 parents 属性。例如:

```
>>> custs.new_child({'X': 'Monica'})
ChainMap({'X': 'Monica'}, {'D': 'Bush', 'B': 'Pika', 'A': 'Justin'}, {'B': 'Irene', 'C': 'caterpillar'})
>>> custs.parents
ChainMap({'B': 'Irene', 'C': 'caterpillar'})
>>>
```


9.2.3 `__getitem__()`、`__setitem__()`、`__delitem__()`

了解了 `collections` 模块中提供的一些高级群集实现之后,接下来的问题是如果想要根据自己的需求来实现群集,该如何进行呢? Python 中其实提供了一些基类,可以让我们基于这些类来实现自己的群集。不过在这之前要先认识 `__getitem__()`、`__setitem__()`、`__delitem__()`。

简单来说,在 Python 的群集中有许多类型都可以使用 `[]` 来指定索引或者是键进行存取,如果想要运用 `[]` 取值,可以实现 `__getitem__()`, 想要运用 `[]` 设值,可以实现 `__setitem__()`, 若想通过 `del` 与 `[]` 来删除,可以实现 `__delitem__()`。

作为示范,下面的范例实现了 `__getitem__()`、`__setitem__()`、`__delitem__()`, 以模仿 `ChainMap` 的部分功能。

collection_advanced chainmap.py

```
class ChainMap:
    def __init__(self, *maps):
        self.maps = maps

    def lookup(self, key):
        for m in self.maps:
            if key in m:
                return m
        return None

    def __getitem__(self, key):
        m = self.lookup(key)
        if m:
            return m[key]
        else:
            raise KeyError(key)

    def __setitem__(self, key, value):
        m = self.lookup(key)
        if m:
            m[key] = value
        else:
            self.maps[key] = value

    def __delitem__(self, key):
        m = self.lookup(key)
        if m:
            del m[key]
        else:
            raise KeyError(key)

c = ChainMap({'A' : 'Justin'}, {'A' : 'Monica', 'B' : 'Irene'})
print(c.maps)

print(c['A'])

c['A'] = 'caterpillar'
print(c.maps)

del c['A']
print(c.maps)
```

① 查找是否有对应键的字典

② 实现 `__getitem__()` 方法

③ 实现 `__setitem__()` 方法

④ 实现 `__delitem__()` 方法

范例中定义了 `lookup()` 方法，可指定键来获取第一个含有指定键的字典，若都没有就返回 `None`^①。`__getitem__()` 的第二个自变量就是 `c[key]` 时的 `key` 值^②，`__setitem__()` 的第二与第三个自变量则是 `c[key] = value` 时的 `key` 与 `value`^③，至于 `__delitem__()` 的第二个自变量则是 `del c[key]` 时的 `key`^④。范例的执行结果如下：

```
{'A': 'Justin'}, {'B': 'Irene', 'A': 'Monica'})
Justin
({'A': 'caterpillar'}, {'B': 'Irene', 'A': 'Monica'})
({}, {'B': 'Irene', 'A': 'Monica'})
```

虽然这里以实现 `ChainMap` 作为示范，然而 `__getitem__()`、`__setitem__()` 与 `__delitem__()` 的第二个自变量也可以是数字，也就是当指定索引时调用这三个方法的具体功能。

附带一提的是，使用 `len()` 函数打算获取一个群集的长度时会调用群集的 `__len__()` 方法，因此，可以在自定义群集时实现 `__len__()` 方法来计算群集的长度并返回长度值。

9.2.4 使用 `collection.abc` 模块

刚才实现的 `ChainMap` 范例其实只是部分模仿了字典的行为。实际上，要实现一个群集对象能够符合 Python 中对群集对象相关的协议要求，还有其他方法必须实现，而且就算记得有哪些方法要实现，自行逐一实现这些方法也是件麻烦且容易出错的任务。

为此，Python 标准链接库提供了 `collections.abc` 模块，`abc` 这名称来看，可以联想到 6.2.5 小节曾介绍过的抽象基类 (Abstract Base Class)，事实上也是如此，`collections.abc` 模块中提供了许多实现群集时的基类，开发者继承这些类可以避免遗忘必须实现的方法，也可以有一些基本的共享实现。

`collections.abc` 模块中的类分类与 9.2.1 小节的介绍相关。`Sequence` 可用来实现序列类型的共同行为，而 `MutableSequence` 继承 `Sequence`，定义了可变动序列类型的行为；`Set` 用来定义集合类型，而 `MutableSet` 继承 `Set`，用来定义可变动集合；`Mapping` 用来定义映射类型，而 `MutableMapping` 继承 `Mapping`，定义了可变动映射类型的行为。

`collections` 模块中的 `ChainMap` 实际上就是继承 `MutableMapping` 而实现的，如果想自行定义 `ChainMap`，除了 `__getitem__()`、`__setitem__()` 与 `__delitem__()` 之外，必要的实现还有 `__iter__()`、`__len__()` 方法，至于 `__contains__()`、`keys()`、`items()`、`values()`、`get()` 等字典的行为，都有默认的 `Mixin` 实现，详细清单可参考 `collections.abc` 模块^①的官方文件。

因此，刚才自行实现的 `ChainMap` 可以改继承 `MutableMapping`，以便符合字典的对象协议。

```
collection_advanced chainmap2.py
```

```
from collections.abc import MutableMapping
```

```
class ChainMap(MutableMapping):
```

```
    def __init__(self, *maps):
```

```
        self.maps = maps
```

```
    def lookup(self, key):
```

^① `collections.abc` 模块：docs.python.org/3/library/collections.abc.html

```

    for m in self.maps:
        if key in m:
            return m
    return None

def keySet(self):
    keys = set()
    for m in self.maps:
        keys.update(m.keys())
    return keys

def __getitem__(self, key):
    m = self.lookup(key)
    if m:
        return m[key]
    else:
        raise KeyError(key)

def __setitem__(self, key, value):
    m = self.lookup(key)
    if m:
        m[key] = value
    else:
        self.maps[key] = value

def __delitem__(self, key):
    m = self.lookup(key)
    if m:
        del m[key]
    else:
        raise KeyError(key)

def __iter__(self):
    return iter(self.keySet())

def __len__(self):
    return len(self.keySet())

c = ChainMap({'A': 'Justin'}, {'A': 'Monica', 'B': 'Irene'})
print(list(c))
print(len(c))
print(c.pop('A'))
print(list(c.keys()))

```

在上面的范例中，继承 `MutableMapping` 后实现了必要的 `__getitem__()`、`__setitem__()`、`__delitem__()`、`__iter__()`、`__len__()` 方法，其他字典的行为（如 `pop()`、`keys()` 等）就自动拥有了，范例的执行如下所示：

```

['A', 'B']
2
Justin
['A', 'B']

```

要留意的是，**Mapping** 并不是字典 (`dict`) 的子类，只是拥有字典的行为，**Sequence** 也不是列表 (`list`) 的子类，只是拥有列表的行为，**Set** 也不是集合 (`set`) 的子类，只是拥有集合的行为。

如果项目规格书或者相关链接库要求群集的相关实现，必须是列表、集合、字典的子类（使用 `isinstance()` 进行判断），那么必须继承列表、集合、字典等来实现，而不是 `Sequence`、`Set`、`Mapping` 等。

提示 >>>

虽然 hashable、iterable、iterator 等对象协议相对来说比较简单，不过 collections.abc 模块中也定义了 Hashable、Iterable、Iterator 类作为对应。

9.2.5 UserList、UserDict、UserString 类

如果只是想要基于字符串、列表、字典等行为（或操作）增加一些自己的方法定义，那么也可以使用 collections 模块的 UserString、UserList、UserDict，它们分别是 Sequence、MutableSequence、MutableMapping 的子类。

举个例子来说，虽然 Python 中不常见到方法链（Method chain）操作，不过下面故意实现了一个可进行方法链操作的 MthChainList 类。

collection_advanced chainable.py

```
from collections import UserList    ← ❶ 继承 UserList 类

class MthChainList(UserList):      ← ❷ 可返回 MthChainList 实例的 filter()方法
    def filter(self, predicate):
        return MthChainList(elem for elem in self if predicate(elem))

    def map(self, mapper):          ← ❸ 可返回 MthChainList 实例的 map()方法
        return MthChainList(mapper(elem) for elem in self)

    def for_each(self, action):     ← ❹ 针对各元素执行指定的操作（或动作）
        for elem in self:
            action(elem)

lt = MthChainList(['a', 'B', 'c', 'd', 'E', 'f', 'G'])
lt.filter(str.islower).map(str.upper).for_each(print)
```

在这里的范例继承了 UserList 之后❶，并没有重新定义父类的任何方法，只是增加了可返回 MthChainList 实例的 filter()方法❷与 map()方法❸，以及可以针对各元素执行指定操作的 for_each()。因为 filter()、map()返回的都是 MthChainList 实例，所以可以直接进行链状操作，在某些程序设计语言中，这样的链状操作是很受欢迎的方式。

提示 >>>

这个范例也可以改为继承列表，若要求 MthChainList 必须是列表的子类，这样做也会更符合需求，实际上 UserList 等类的存在还有着过去版本的 Python 不允许直接继承列表等内建类的历史渊源。

9.3 重点复习

一个对象能被称为 hashable（可哈希运算的），它必须有 hash 值（哈希值），这个值在整个运行时刻都不会变化，而且必须可以进行相等比较。具体来说，一个对象能被称为 hashable，它必须实现 __hash__()与 __eq__()方法。

对于 Python 内建类型来说，只要是创建后状态就无法变动的类型，它的实例都是 `hashable`，可变动的类型的实例都是 `unhashable`。

一个自定义的类创建的实例默认也是 `hashable` 的，其 `__hash__()` 的实现基本上是根据 `id()` 计算而来，`__eq__()` 的实现默认是使用 `is` 来比较。因此，两个分别创建的实例 `hash` 值必然不相同，而且相等性对比一定不成立。

`hashable` 对象建议状态不可变动。两个对象若是相等性比较成立，那么也必须有相同的 `hash` 值，然而 `hash` 值相同，两个对象的相等性比较不一定是成立的。

具有 `__iter__()` 方法的对象就是一个 `iterable`（可迭代的）对象。迭代器具有 `__next__()` 方法，可以逐一迭代出对象中的信息，若无法进一步迭代，会引发 `StopIteration`。迭代器也会具有 `__iter__()` 方法返回迭代器自身，因此每个迭代器本身也是个 `iterable` 对象。

在 Python 标准链接库中提供了 `itertools` 模块，当中有许多函数可协助创建迭代器或生成器。

列表才有 `sort()` 方法，对于其他 `iterable` 对象，若想进行排序，可以使用 `sorted()` 函数，可指定的参数同样也有 `reverse` 与 `key` 参数。此函数不会变动原有的函数，排序的结果会以新的列表返回。

在 Python 中，大致上将群集分为三种类型：序列类型、集合类型与映射类型。

不可变动的序列类型具有默认的 `hash()` 实现。

集合类型是无序的，而且元素必须都是 `hashable` 对象而且不会重复，它们也是 `iterable` 对象，可以使用 `x in set`、`x not in set`、`len(set)` 以及交集、并集、差集与对称差集等操作。

集合本身是可变动的，如果想要不可变动的集合类型，可以使用 `frozenset()` 来创建，创建的实例本身实现了 `__hash__()` 方法，为 `hashable` 对象。

对于队列或双向队列来说，使用列表的效率并不好，对于队列或双向队列的需求，建议使用 `collections` 模块中提供的 `deque` 类。

如果想要有个简单类，以便创建的实例能拥有字段名，实际上不用自行定义，可以使用 `collections` 模块的 `namedtuple()` 函数。

如果想要在创建字典时保有最初键和值加入的顺序，可以使用 `collections` 模块的 `OrderedDict`。`defaultdict` 接受一个函数，它创建的实例在指定的键不存在时就会使用指定的函数来产生，并直接设置键的对应值。

如果想要运用 `[]` 取值，可以实现 `__getitem__()`，想要运用 `[]` 设值，可以实现 `__setitem__()`，若想通过 `del` 与 `[]` 来删除，可以实现 `__delitem__()`。

可以在自定义群集时实现 `__len__()` 方法来计算群集的长度并返回长度值。

`collections.abc` 模块中提供了许多实现群集时的基类，开发者继承这些类可以避免遗忘必须实现的方法，也可以有一些基本的共享实现。

`Mapping` 并不是字典 (`dict`) 的子类，只是拥有字典的行为，`Sequence` 也不是列表 (`list`) 的子类，只是拥有列表的行为，`Set` 也不是集合 (`set`) 的子类，只是拥有集合的行为。

课后练习

实践题

1. 尝试编写一个 `MultiMap` 类，行为上像个字典，不过若指定的键已存在，值将会存储在一个集合中，而不是直接覆盖字典中现有的对应值。例如要有以下的行为：

```
mmap = MultiMap({'A' : 'Justin'}, {'A' : 'Monica', 'B' : 'Irene'})
print(mmap) # 显示 {'B': {'Irene'}, 'A': {'Justin', 'Monica'}}

mmap['B'] = 'Pika'
print(mmap) # 显示 {'B': {'Irene', 'Pika'}, 'A': {'Justin', 'Monica'}}
```

2. 如果有一个字符串数组如下：

```
words = ['RADAR', 'WARTER START', 'MILK KLIM', 'RESERVED', 'IWI', 'ABBA']
```

请编写程序，判断字符串数组中有哪些字符串，从前面看的字符顺序与从后面看的字符顺序是相同的。

提示 >>>

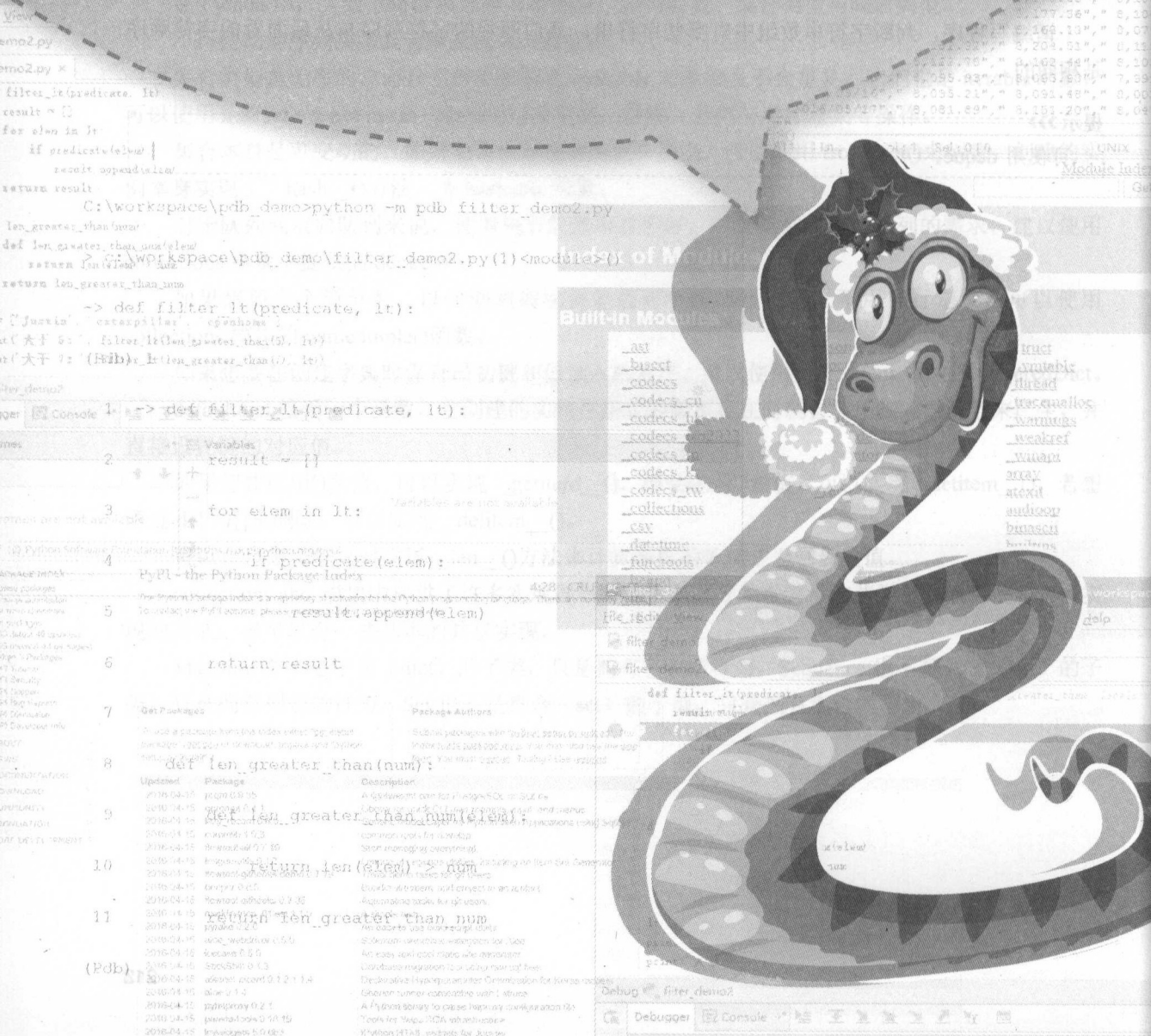
可使用 `deque`。

第 10 章

数据持续性与交换

学习目标

- 使用 pickle 与 shelve
- 认识 DB-API 2.0
- 使用 sqlite3 模块
- 处理 CSV、JSON、XML



10.1 对象序列化

程序运算的结果经常必须保存下来，下次程序运算时就能再度运用，或者传递给另一个程序继续运算，这类能保留程序运算结果的机制称为持久性（Persistence）。程序运行时能将内存中的对象信息直接保存下来的机制通常称为对象序列化（Object serialization），反之若将保存下来的数据读取并转换为存储区中的对象信息，称之为反序列化（Deserialization）。

10.1.1 使用 pickle 模块

如果要序列化 Python 对象可以使用内建的 pickle 模块，它能记录已经序列化的对象，如果后续有对象引用到相同对象才不会再度被序列化。

使用 pickle 进行对象序列化时会将一个 Python 对象转换为字节流（bytes），在 Python 的术语中，这个过程称为 Pickling（英文原意为“腌制”），相反的操作则称之为 unpickling，会将字节流（bytes）转换为 Python 对象。Python 使用 pickling、unpickling 来称呼是为了避免与 serialization（串行化）、marshalling（编组）等类似的名词混淆。

在 pickle 模块的使用上，若想将对象转换为字节流（bytes），可以使用 dumps() 函数，若想将一个代表对象的字节流（bytes）转换为对象，可以使用 loads() 函数。例如：

```
>>> import pickle
>>> custs = {'A': ('Justin', [10, 20]), 'B': ('Monica', [30, 40])}
>>> pickled = pickle.dumps(custs)
>>> pickled
b'\x80\x03q\x00(X\x01\x00\x00\x00Aq\x01X\x06\x00\x00\x00Justinq\x02]q\x03(K\nK\x14e\x86q\x04X\x01\x00\x00\x00Bq\x05X\x06\x00\x00\x00Monicaq\x06]q\x07(K\x1eK(e\x86q\x08u.'
>>> pickle.loads(pickled)
{'A': ('Justin', [10, 20]), 'B': ('Monica', [30, 40])}
>>>
```

在这个例子中，pickled 引用的是字节流（bytes），这时可以将之传送到另一个目的地，也许是网络的另一端，或者是文件，另一方收到字节流（bytes）之后，也就可以使用 loads() 转换回 Python 对象。

注意 >>>

在 pickle 模块的官方帮助文档中，一开始就有个显眼的 Warning，警告绝对不要从不信任的源执行 unpickling 的操作，因为可能会含有恶意的字节信息。

可以 pickling 与 unpickling 的类型¹包括了 Python 内建类型、用户自定义的顶层函数、类等，如果无法进行 pickling 或 unpickling，就会引发 PicklingError 错误或 UnpicklingError 错误（父类都是 PickleError）。

¹ What can be pickled and unpickled?: docs.python.org/3.5/library/pickle.html#pickle.dump

如果 pickling 之后想要直接将字节流 (bytes) 保存在文件中, 可以使用 dump() 函数, 它有个 file 参数可以指定文件对象, 文件对象必须是二进制模式。如果 unpickling 的来源是个文件, 可以使用 load() 来读取并转为 Python 对象, 它也有个 file 参数可以指定文件对象, 文件对象必须是二进制模式。下面是个简单的范例:

```
>>> import pickle
>>> custs = {'A': ('Justin', [10, 20]), 'B': ('Monica', [30, 40])}
>>> with open('custs.pickle', 'wb') as f:
...     pickle.dump(custs, file=f)
...
>>> with open('custs.pickle', 'rb') as f:
...     pickle.load(file=f)
...
{'A': ('Justin', [10, 20]), 'B': ('Monica', [30, 40])}
>>>
```

来看一个更实际的使用 pickle 的程序范例, 这个范例也示范了实现持续性机制的一种模式, 用来保存 DVD 对象的状态。



object_serialization dvdlib_pickle.py

```
import pickle

class DVD:
    def __init__(self, title, year=None, duration=None, director_id=None):
        self.title = title
        self.year = year
        self.duration = duration
        self.director_id = director_id
        self.filename = self.title.replace(' ', '_') + '.pickle' ① 存储的文件名

    def save(self, filename=None): ② 存储对象
        with open(self.filename, 'wb') as fh:
            pickle.dump(self, fh)

    @staticmethod
    def load(filename): ③ 读取文件获取对象
        with open(filename, 'rb') as fh:
            return pickle.load(fh)

    def __str__(self):
        return repr(self)

    def __repr__(self):
        return "DVD('{0}', {1}, {2}, '{3}').format(
            self.title, self.year, self.duration, self.director_id)

dvd1 = DVD('Birds', 2016, 1, 'Justin Lin')
dvd1.save()
dvd2 = DVD.load('Birds.pickle')
print(dvd2) #显示DVD('Birds', 2016, 1, 'Justin Lin')
```

这个 DVD 对象有 title、year、duration、director_id 4 个状态, 每个 DVD 对象会以 title 作为主文件名, 空白用下划线取代, 并加上 .pickle 扩展名进行存盘①。在存储对象的 save() 方法中, 使用 'wb' 模式打开文件, 然后使用 pickle.dump() 进行 pickling②。至于 unpickling 的 load() 方法, 在这里设计为静态方法, 使用 'rb' 模式打开文件, 可指定文件名加载并获取 DVD 对象③。

pickling 时实际采用的模式是 Python 的专用格式, pickle 的保证是能向后兼容未来的新版本。格式历经几个版本的更迭, 版本 2 是在 Python 2.3 时导入的, 版本 3 是在 Python 3.0 时导入

的，编写本书时最新的版本是版本 4，是在 Python 3.4 时导入的。

可以使用 `pickle.HIGHEST_PROTOCOL` 来得知当前可用的最新格式版本为哪一个，而 `pickle.DEFAULT_PROTOCOL` 是 `pickle` 模块的默认版本，为了整个 Python 3 系列的兼容性，Python 3.4、3.5 的 `pickle.HIGHEST_PROTOCOL` 虽然是 4，不过 `pickle.DEFAULT_PROTOCOL` 值是 3。如果需要指定格式版本，可以在使用 `dumps()`、`dump()`、`loads()` 或 `load()` 时指定其 `protocol` 参数。

提示 >>>

`cPickle` 模块是用 C 语言实现的模块，接口上与 `pickle` 相同，速度在理想上可达 `pickle` 的 1000 倍。不过，并非每个平台上的 Python 都有 `cPickle`（Windows 上就没有），可以使用以下方式尝试使用 `cPickle`，若没有，就使用 `pickle`。

```
try:
    import cPickle except ImportError:
    import pickle
```

10.1.2 使用 shelve 模块

`shelve` 对象行为上像是字典的对象，键的部分必须是字符串，值的部分可以是 `pickle` 模块可处理的 Python 对象，它直接与一个文件关联，因此使用上就像一个简单的数据库接口。现在来看看基本的使用方式：

```
>>> import shelve
>>> dvdlib=shelve.open('dvdlib.shelve')
>>> dvdlib['Birds'] = (2016, 1, 'Justin Lin')
>>> dvdlib['Dogs'] = (2016, 7, 'Monica Huang')
>>> dvdlib.close()
>>> dvdlib=shelve.open('dvdlib.shelve')
>>> dvdlib['Dogs']
(2016, 7, 'Monica Huang')
>>> del dvdlib['Dogs']
>>> dvdlib.sync()
>>> dvdlib['Mouses'] = (2016, 3, 'Irene Lin')
>>> list(dvdlib) ['Mouses', 'Birds']
>>> dvdlib.close()
>>>
```

在这个范例中可以看到，我们可以将文件当成简单的数据库，只要对 `shelve.open()` 创建的对象进行字典（dict）一样的操作，在 `close()` 或 `sync()` 时数据就会存储到文件中。

提示 >>>

`shelve` 的底层使用 `dbm`，`dbm` 为伯克利大学开发的文件型数据库，Python 的 `dbm` 模块提供了对 Unix 链接库的接口；由于底层使用 `dbm`，因此功能上也会受到 `dbm` 模块 `shelve` 的限制¹。

¹ `shelve` 的限制：docs.python.org/3.5/library/shelve.html#restrictions

下面示范了另一种模式，来封装 `shelve` 的操作行为。



object_serialization dvdlib_shelve.py

```
import shelve

class DVD:
    def init (self, title, year=None, duration=None, director_id=None):
        self.title = title
        self.year = year
        self.duration = duration
        self.director_id = director_id

    def str (self):
        return repr(self)

    def repr (self):
        return ("DVD('{title}', {year}, {duration}, '{director_id}')"
                .format(**vars(self)))

class DvdDao:
    def init (self, dbname):
        self.dbname = dbname

    def save(self, dvd):
        with shelve.open(self.dbname) as shelve_db:
            shelve_db[dvd.title] = dvd
        ← ①存储 DVD 对象

    def all(self):
        ← ②获取全部 DVD 对象，按标题小写字母排序后返回
        with shelve.open(self.dbname) as shelve_db:
            shelve_db = shelve.open(self.dbname)
            return (shelve_db[title]
                    for title in sorted(shelve_db, key = str.lower))

    def load(self, title):
        ← ③指定标题返回 DVD 对象
        with shelve.open(self.dbname) as shelve_db:
            if title in shelve_db:
                return shelve_db[title]
            return None

    def remove(self, title):
        ← ④指定标题删除 DVD 对象
        with shelve.open(self.dbname) as shelve_db:
            del shelve_db[title]

dao = DvdDao('dvdlib.shelve')
dvd1 = DVD('Birds', 2016, 1, 'Justin Lin')
dvd2 = DVD('Dogs', 2016, 7, 'Monica Huang')
dao.save(dvd1)
dao.save(dvd2)
print(list(dao.all()))
print(dao.load('Birds'))
dao.remove('Birds')
print(list(dao.all()))
```

在 `save()` 方法中，主要是使用 `shelve.open()` 来打开文件。在指定键值之后，`with` 自动关闭文件之前会将数据从缓存中写回文件①，在 `all()` 方法中，打开文件读取并获取 `shelve` 的对象之后，将全部 DVD 对象按小写字母排序后，返回一个生成器②。`load()` 方法可以指定标题返回到 DVD 对象③，而 `remove()` 方法可以指定标题删除 DVD 对象④。

注意>>>

由于 `shelve` 是以 `pickle` 为基础，因此绝对不要读取不信任的文件，因为可能含有恶意的字节信息。

10.2 数据库的处理

对于关系数据库 (Relational database) 的存取，在 Python 中的标准规范是 DB-API 2.0，标准链接库内建的 `sqlite3` 模块就符合此规范，`SQLite` 是一个轻量级数据库，用来学习数据库的处理或满足基本需求都非常的方便。

10.2.1 认识DB-API 2.0

DB-API 2.0 遵循PEP249 规范¹，所有的数据库接口都应该符合这个规范，以便编写程序时能有一致的方式，编写出来的程序也便于跨数据库执行。实际上模块在实现时可能提供更多的功能。

在数据库的连接上，DB-API 2.0 规范数据库模块实现时必须提供 `connect(parameters...)` 函数，用以构建 `Connection` 对象，`Connection` 基本上要具备表 10.1 所示的方法。

表 10.1 `Connection` 的基本方法

方法	说明
<code>close()</code>	关闭当前的数据库连接
<code>commit()</code>	将尚未完成的交易提交
<code>rollback()</code>	将尚未完成的交易撤回
<code>cursor([cursorClass])</code>	返回一个 <code>Cursor</code> 对象，代表基于当前连接的数据库游标，所有和数据库的会话都通过 <code>Cursor</code> 对象

使用 `Connection` 的 `cursor()` 方法创建的 `Cursor` 对象可以用来执行 SQL 语句，在 DB-API 2.0 的规范中，`Cursor` 对象基本上必须具备表 10.2 所示的方法。

表 10.2 `Cursor` 的基本方法

方法	说明
<code>close()</code>	关闭当前的 <code>Cursor</code> 对象
<code>execute(sql[, params])</code>	执行一次 <code>sql</code> 语句，可以是查询 (Query) 或发出指令 (Command)
<code>executemany(sql, seq_of_params)</code>	针对 <code>seq_of_params</code> 序列或映射中每项执行一次 <code>sql</code> 语句
<code>fetchone()</code>	从查询的结果集中获取下一笔数据
<code>fetchmany([size])</code>	从查询的结果集中获取多笔数据
<code>fetchall()</code>	从查询的结果集中获取全部数据

`Cursor` 对象本身也有一些属性可获得数据的相关信息。例如，`description` 属性会是一个序列，

¹ PEP 249: www.python.org/dev/peps/pep-0249/

里头每个元素为(name、type_code、display_size、internal_size、precision、scale、null_ok)，也就是字段的 7 个信息；rowcount 表示 execute() 执行 SQL 之后影响了多少笔数据；arraysize 决定了 fetchmany() 方法默认会取回多少笔数据。

由于各个数据库产品都有不同的特性，学习 Python 如何进行数据库连接可以从这些 DB-API 2.0 规范的基本方法与属性开始，若想知道 Python 当前支持的数据库接口以及各自的特性，可以查阅 “DatabaseInterfaces¹” 中的说明。

10.2.2 使用 sqlite3 模块

若想马上在 Python 中进行数据库程序的编写，我们不需要特别下载、安装以及创建数据库服务器，Python 中内建了 SQLite 数据库，这是一个用 C 语言编写的轻量级数据库，数据库本身的数据可以存储在一个文件或者内存中，后者对于数据库应用程序的测试非常方便。

若想使用 SQLite 作为数据库，并编写 Python 程序与数据库进行操作，可以使用 sqlite3 模块，这个模块遵循 DB-API 2.0 的规范而实现。下面先就基本的数据库创建数据表、数据新增、查询、更新与删除进行示范。

创建数据库与连接数据库

想要创建一个数据库文件，可以使用 sqlite3.connect() 函数并指定文件名称，如果数据库文件尚不存在就会创建一个新的文件并打开数据库连接，如果文件存在就直接打开连接，并返回一个 Connection 对象。例如：

```
>>> import sqlite3
>>> conn=sqlite3.connect('db.sqlite3')
>>> conn
<sqlite3.Connectionobject at 0x00B57F00>
>>> conn.close()
>>>
```

如果是首次执行以上的范例，工作目录下就会出现 db.sqlite3 文件，也可以传给 connect() 一个 'memory:' 字符串，这样就会在内存中创建一个数据库。在不使用数据库的时候，应该调用 Connection 的 close() 关闭连接，以释放数据库连接的相关资源。

创建数据表与新增数据

如果想要在数据库中新增数据表，可以使用 Connection 对象的 cursor() 方法获取 Cursor 对象，利用它的 execute() 方法来执行创建数据表的 SQL 语句。例如：

```
>>> conn=sqlite3.connect('db.sqlite3')
>>>
>>> c = conn.cursor()
>>> c.execute(''CREATE TABLE messages (
...     id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,
...     name TEXT NOT NULL,
```

¹ DatabaseInterfaces: wiki.python.org/moin/DatabaseInterfaces

```

...     email TEXT NOT NULL,
...     msg TEXT NOT NULL
... )'''
<sqlite3.Cursor object at 0x00C6E4A0>
>>> conn.commit()
>>> conn.close()
>>>

```

在上面的范例中创建了一个 `messages` 数据表，其中有 `id`、`name`、`email` 与 `msg` 4 个字段，`id` 会自动以流水号方式递增字段值，`sqlite3` 模块的实现默认不会自动提交 SQL 执行后的变更，必须自行调用 `Connection` 的 `commit()` 方法变更才会生效。

不过，由于 `Connection` 对象实现了 7.2.3 小节介绍过的上下文管理器，因此可以搭配 `with` 语句，在 `with` 区块的操作完成之后会自动 `commit()` 与 `close()`。若发生例外，则会自动 `rollback()`。例如，上面的范例也可以改写为以下方式：

```

import sqlite3

with sqlite3.connect('db.sqlite3') as conn:
    c = conn.cursor()
    c.execute('''CREATE TABLE messages (
id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, name TEXT NOT NULL,
email TEXT NOT NULL, msg TEXT NOT NULL
)''')

```

若要新增一笔数据，也是使用 `Cursor` 的 `execute()` 方法，下面的范例直接搭配 `with`。

```

>>> with sqlite3.connect('db.sqlite3') as conn:
...     c = conn.cursor()
...     c.execute("INSERT INTO messages VALUES(1, 'justin', 'caterpillar@openhome.cc', 'message...')")
...
<sqlite3.Cursor object at 0x00C6E4A0>
>>>

```

查询数据

如果要查询数据，可以先用 `Cursor` 的 `execute()` 执行查询语句，使用 `fetchone()` 可以获取结果集中的一笔数据，`fetchall()` 则可获取结果集中的全部数据，或者使用 `fetchmany()` 指定要从结果集中获取几笔数据。例如查询当前 `messages` 数据表中全部的数据（当前只有一笔）。

```

>>> conn = sqlite3.connect('db.sqlite3')
>>> c = conn.cursor()
>>> c.execute('SELECT * FROM messages')
<sqlite3.Cursor object at 0x00C44BA0>
>>> c.fetchall()
[(1, 'justin', 'caterpillar@openhome.cc', 'message...')]
>>> conn.close()
>>>

```

实际上，`Cursor` 本身是个迭代器，每一次的迭代会调用 `Cursor` 的 `fetchone()` 方法，因此若想逐笔迭代结果的集合，也可以使用 `for in` 语句。从上面的执行结果也可以看出，查询得到的每一笔数据会以元组（tuple）方式返回。

更新与删除数据

若想更新数据表中的字段或者删除某几笔数据，都是使用 `Cursor` 的 `execute()` 方法。例如：

```

>>> with sqlite3.connect('db.sqlite3') as conn:

```

```

... c = conn.cursor()
... c.execute("UPDATE messages SET name='Justin Lin' WHERE id = 1")
...
<sqlite3.Cursor object at 0x00EB4BA0>
>>>withsqlite3.connect('db.sqlite3') as conn:
... c = conn.cursor()
... print(list(c.execute("SELECT * FROM messages")))
...
[(1, 'Justin Lin', 'caterpillar@openhome.cc', 'message...')]
>>>withsqlite3.connect('db.sqlite3') as conn:
... c = conn.cursor()
... c.execute("DELETE FROM messages WHERE id = 1")
...
<sqlite3.Cursor object at 0x00EDE560>
>>>withsqlite3.connect('db.sqlite3') as conn:
... c = conn.cursor()
... print(list(c.execute("SELECT * FROM messages")))
... []
>>>

```

由于 `execute()` 执行过后都是返回 `Cursor`，而 `Cursor` 本身是迭代器，因此在上面的范例中直接使用 `list()` 将每一笔数据放在列表中，最后使用 `print()` 来显示。

10.2.3 参数化SQL 语句

在之前的范例中，SQL 中固定“写死”了 `name`、`email`、`msg` 等字段的信息。实际上这些信息可能来自于用户的输入，而我们必须将输入组合为 SQL，再交由 `Cursor` 的 `execute()` 方法执行。

不过，直接使用 `+` 来串接字符串以组成 SQL 容易引发 SQL Injection 的安全问题。举个例子来说，如果原先使用串接字符串的方式来执行 SQL：

```

c = conn.cursor()
query_sql = ("SELECT * FROM user_table WHERE username='" +
             username + "' AND password='" + password + "'")
c.execute(query_sql)

```

其中 `username` 与 `password` 若是来自用户输入的字符串，原本希望用户安分地输入名称和密码，组合之后的 SQL 应该像是这样的：

```

SELECT * FROM user_table
WHERE username='caterpillar' AND password='openhome'

```

但是，如果用户在密码的部分输入了“`OR '1'='1'`”，而我们又没有针对用户的输入进行字符检查和过滤操作，这个奇怪的字符串最后组合出来的 SQL 会如下所示：

```

SELECT * FROM user_table
WHERE username='caterpillar' AND password='OR '1'='1'

```

方框是密码请求参数的部分，将方框拿掉会更清楚地看出这个 SQL 有什么问题。

```

SELECT * FROM user_table
WHERE username='caterpillar' AND password='OR '1'='1'

```

AND 子句之后的判断式永远成立，也就是说，用户不用输入正确的密码也可以查询出所有数

据, 这就是 **SQL Injection** 的简单例子。

读者也许会想到, 使用字符串的 `format()` 或者旧式的 `%` 进行格式化。不过, 这也会有同样的问题, 因为它们也是将指定的字符串直接拿来组合为 SQL 语句, 不会进行任何转义 (Escape) 的操作。

```
>>> username = 'caterpillar'
>>> password = "' OR '1'='1"
>>> "SELECT * FROM user_table WHERE username='{}' AND password = '{}'.format(username, password)
"SELECT * FROM user_table WHERE username='caterpillar' AND password=''' OR '1'='1'"
>>> "SELECT * FROM user_table WHERE username='%s' AND password = '%s'" % (username, password)
"SELECT * FROM user_table WHERE username='caterpillar' AND password=''' OR '1'='1'"
>>>
```

Cursor 的 `execute()` 方法本身可以将 SQL 语句参数化, 有两种参数化的方式: 使用问号 (?) 或具名占位符号。例如使用问号作为占位符号:

```
c = conn.cursor()
query_sql = "SELECT * FROM user_table WHERE username=? AND password=?"
c.execute(query_sql, (username, password))
```

`execute()` 的第一个参数指定了包含占位符号的字符串, 第二个参数指定了一个元组, 元素顺序对应于占位符号的顺序, 元素会经过转义而不是直接拿来取代字符串, 这样就可以避免刚才谈到的 SQL Injection 问题。下面是使用具名占位符号的例子:

```
c = conn.cursor()
query_sql = "SELECT * FROM user_table WHERE username=:username AND password=:password"
c.execute(query_sql, {'username': username, 'password': password})
```

可以看到, 使用具名占位符号时, 必须加上冒号 (:) 作为前导字符, 而在 `execute()` 的第二个参数上, 是使用字典来指定实际数据的。如果有多笔 SQL 必须执行, 虽然可以使用 `for in` 自行处理:

```
messages = [
    (1, 'Justin Lin', 'caterpillar@openhome.cc', 'message1...'), (2, 'Monica Huang',
    'monica@openhome.cc', 'message2...'),
    (3, 'IreneLin', 'irene@openhome.cc', 'message3...')
]
for message in messages:
    c.execute("INSERT INTO messages VALUES (?, ?, ?, ?)", message)
```

但是使用 **Cursor** 的 `executemany()` 会更方便。

```
messages = [
    (1, 'Justin Lin', 'caterpillar@openhome.cc', 'message1...'), (2, 'Monica Huang',
    'monica@openhome.cc', 'message2...'),
    (3, 'IreneLin', 'irene@openhome.cc', 'message3...')
]
c.executemany("INSERT INTO messages VALUES (?, ?, ?, ?)", messages)
```

除了使用问号之外, **Cursor** 的 `executemany()` 也可以使用具名占位符号。

10.2.4 简介交易

交易的 4 个基本要求是原子性 (**Atomicity**)、一致性 (**Consistency**)、隔离行为 (**Isolation**)

behavior) 与持久性 (Durability), 按英文单词首字母就简称为 ACID。

- 原子性

一个交易是一个单元工作 (Unit of work), 当中可能包括数个步骤, 这些步骤必须全部执行成功, 若有一个失败, 则整个交易声明失败, 交易中其他步骤必须撤销已执行过的操作, 回到交易之前的状态。

在数据库上执行单元工作进行数据库交易 (Database transaction), 单元中每个步骤就是每一句 SQL 的执行, 我们要定义开始一个交易的边界 (通常是以一个 BEGIN 的指令开始), 所有 SQL 语句下达之后, COMMIT 确认所有操作变更, 此时交易成功, 或者因为某个 SQL 错误, 用 ROLLBACK 进行撤销操作, 此时交易失败。

- 一致性

交易作用的数据集合在交易前后必须一致, 若交易成功, 整个数据集合都必须是交易操作后的状态。若交易失败, 整个数据集合必须与开始交易前一样没有变更, 不能发生整个数据集合部分有变更, 而部分没变更的状态。

例如转账行为, 数据集合涉及 A、B 两个账户, A 原有 20000, B 原有 10000, A 转 10000 给 B, 如果交易成功, 那么最后 A 必须变成 10000, B 变成 20000。如果交易失败, 那么 A 必须为 20000, B 为 10000, 不能发生 A 为 20000 (未扣款) B 也为 20000 (已入款) 的情况。

- 隔离性

在多人使用的环境下, 每个用户可能进行自己的交易, 交易与交易之间必须互不干扰, 用户不会意识到别的用户正在进行交易, 就好像只有自己在进行操作一样。

- 持久性

交易一旦成功, 所有变更必须保存下来, 即使系统“挂了”, 交易的结果也不能遗失, 这通常需要系统软、硬件架构的支持。

在原子性的处理上, sqlite3 模块会在 INSERT、UPDATE、DELETE、REPLACE 等会变更数据的 SQL 操作前隐含地启动交易, 在任何非变更数据的 SQL 操作及一些 CREATE 数据表等其他情况下隐含地进行提交。

除了一些会隐含提交的情况之外, sqlite3 模块的默认实现并不会自动提交, 因此我们必须自行调用 Connection 的 commit() 进行提交, 如果交易过程因为发生错误或其他情况必须撤回交易, 可以调用 Connection 的 rollback() 执行撤回操作。

一个基于例外发生时必须撤销交易的示范如下:

```
conn = None
try:
    conn = sqlite.connect('example.sqlite')
    c = conn.cursor()
    c.execute("INSERT INTO...")
    c.execute("INSERT INTO...")
    conn.commit() #提交
except DatabaseError as e:
    #做一些日志记录
```

```
if conn:
    conn.rollback() #撤回
```

在 10.2.2 小节谈过了, Connection 对象实现了 7.2.3 小节介绍过的上下文管理器, 因此可以搭配使用 with 语句, 在 with 区块的操作完成之后会自动执行 commit() 与 close(), 若发生例外, 则会自动 rollback()。

在隔离性方面, SQLite 数据库在更新数据的相关操作时默认会锁定数据库直到该次交易完成, 因此在有多个连接时会造成等待的情况, sqlite3 模块的 connect()函数有个 timeout 可指定等待多久, 如果超时就引发例外, 默认是 5.0, 也就是 5 秒。

sqlite3 模块的 Connection 对象有个 isolation_level 属性, 可用来设置或得知当前的隔离性设置, 默认是", 实际上在 SQLite 数据库会产生 BEGIN 语句, 如果 isolation_level 被设置为 None, 表示不进行任何隔离, 也就成为自动提交, 每次 SQL 更新相关操作时就不用自行调用 Connection 的 commit()方法。

然而, 不进行隔离在多个连接访问数据库的情况下会引发数据不一致的问题, 下面逐一举例说明。

更新丢失 (Lost update)

基本上是指某个交易对字段进行更新的信息因另一个交易的介入丢失了。举例来说, 若某个字段数据原来为 ZZZ, 用户 A、B 分别在不同的时间点对同一字段进行更新交易, 过程如图 10-1 所示。

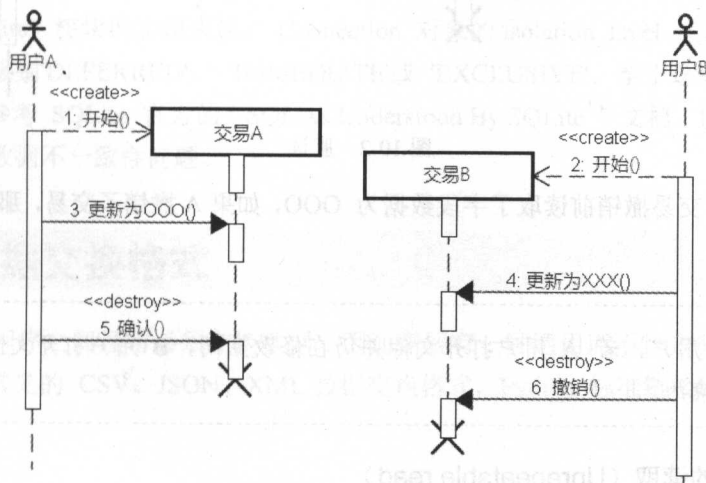


图 10-1 更新丢失

单就用户 A 的交易而言, 最后字段应该是 OOO, 单就用户 B 的交易而言, 最后字段应该是 ZZZ, 在完全没有隔离两者交易的情况下, 由于用户 B 撤销操作的时间在用户 A 确认之后, 最后字段结果是 ZZZ, 用户 A 看不到他自己更新确认的 OOO 结果, 就是用户 A 发生了更新丢失的问题。

提示>>>

可以想象有两个用户，若 A 用户打开文件之后，又允许 B 用户打开文件，一开始 A、B 用户看到的文件都有 ZZZ 文字，A 修改 ZZZ 为 OOO 后保存，B 修改 ZZZ 为 XXX 后又还原为 ZZZ 并保存，最后文件内容为 ZZZ，A 用户的更新丢失了。

❶ 脏读（Dirty read）

两个交易同时进行，其中一个交易更新数据但未确认，另一个交易就读取数据，就有可能发生脏读问题，就是读到脏数据（Dirty data），也就是不干净、不正确的数据，如图 10-2 所示。

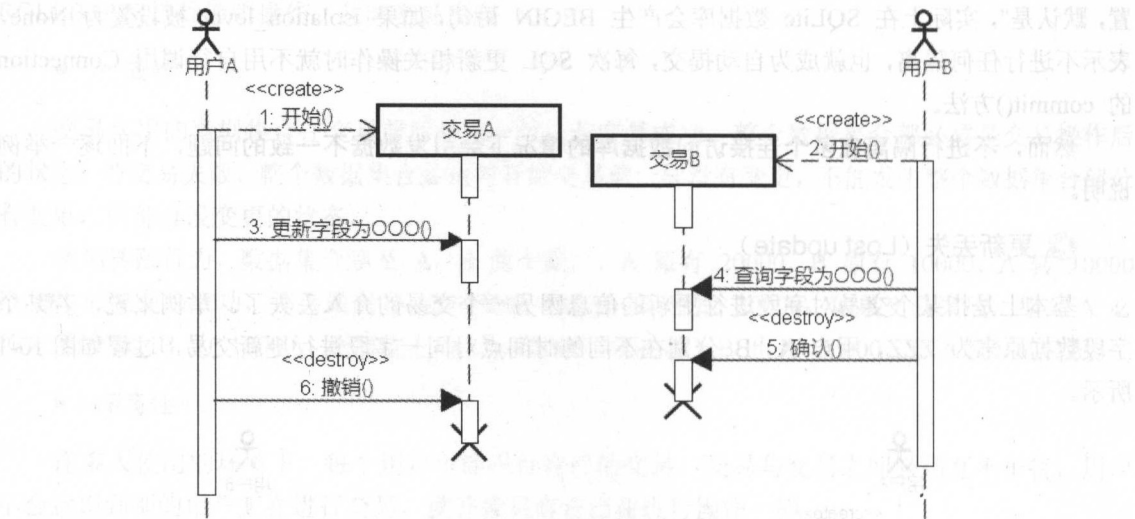


图 10-2 脏读

用户 B 在 A 交易撤销前读取了字段数据为 OOO，如果 A 撤销了交易，那用户 B 读取的数据就是不正确的。

提示>>>

可以想象有两个用户，若 A 用户打开文件并仍在修改期间，B 用户打开文件所读到的数据就有可能是不正确的。

❷ 无法重复的读取（Unrepeatable read）

某个交易两次读取同一字段的数据并不一致。例如，交易 A 在交易 B 更新前后进行数据的读取，则 A 交易会得到不同的结果，如图 10-3 所示（若字段原先为 ZZZ）。

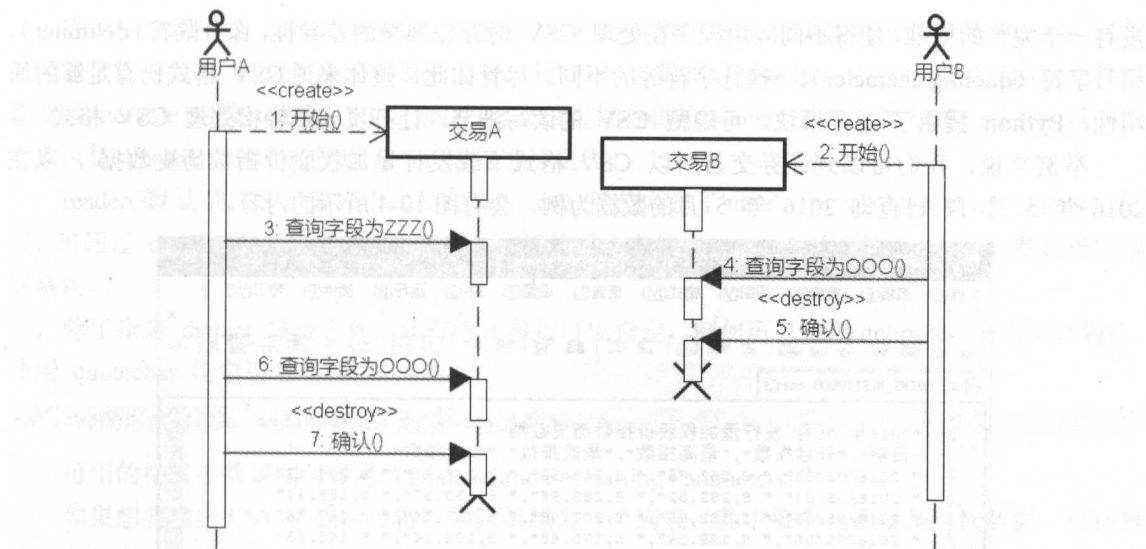


图 10-3 无法重复的读取 (Unrepeatable read)

幻读 (Phantom read)

同一交易期间读取到的数据笔数不一致。例如交易 A 第一次读获取 5 笔数据, 此时交易 B 新增了一笔数据, 导致交易 B 再次读获取到 6 笔数据。

由于各家数据库对于交易的支持程度并不相同, 实际上该采用哪家的以及如何设置也就有所差异, 就 sqlite3 模块的实现来说, Connection 对象的 isolation_level 还可以设置 SQLite 数据库支持的隔离级别 'DEFERRED'、'IMMEDIATE' 或 'EXCLUSIVE'。至于这些隔离级别设置的作用, 详细信息可参考 SQLite 官方的“SQL As Understood By SQLite¹”文档, 以便了解各个设置能够预防什么样的数据不一致性问题。

10.3 数据交换格式

不同的应用程序之间经常必须交换数据, 因而需要有一种通用而不特定应用程序专用的数据交换格式, 对于常见的 CSV、JSON、XML 数据交换格式, Python 标准链接库都内建了对应的处理模块。

10.3.1 CSV

CSV 的全名为 **Comma Separated Values** (逗号分隔值), 是一种通用于电子表格、数据库之间的数据交换格式。实际上在 RFC4180² 试图为其制订标准之前, CSV 已通用多年, 由于多年来

¹ SQL As Understood By SQLite: www.sqlite.org/lang_transaction.html

² RFC4180: tools.ietf.org/html/rfc4180.html

没有一个完善的标准,使得不同应用程序在处理 CSV 时存在细微的差异性,像分隔符(delimiter)、引号字符(quoting character)、换行字符等的不同。尽管如此,整体来说 CSV 格式仍有足够的通用性,Python 提供了 csv 模块,可隐藏 CSV 的读写细节,让开发人员轻松处理 CSV 格式。

举例来说,我们可以到证券交易所以 CSV 格式下载发行量加权股价指数历史数据¹,以在 2016 年 5 月 17 日查询 2016 年 5 月的数据为例,会有图 10-4 所示的内容。

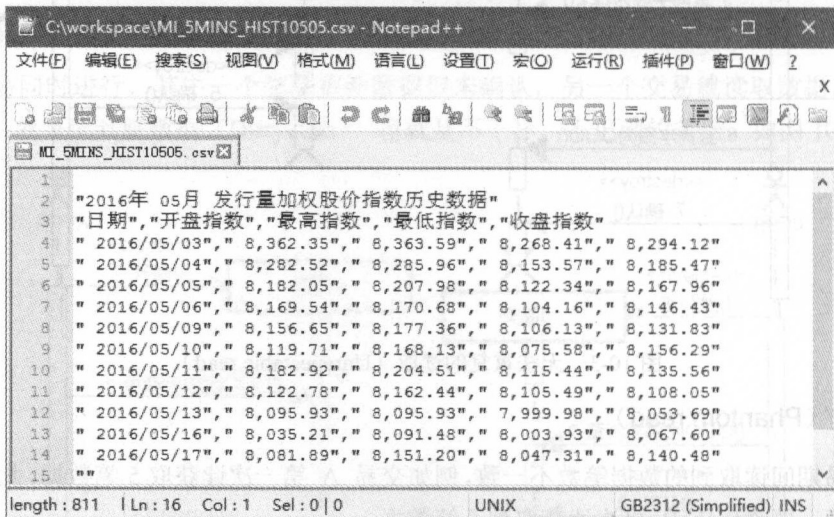


图 10-4 CSV 文件范例

使用 reader()、writer()

这个 CSV 文件的编码是 GB2312,如果简单地使用逗号(,),对每一行的字段进行分割就会很麻烦,因为有些字段中也有逗号,如果使用 Python 的 csv 模块就可以轻松读取。例如:

```
>>> import csv
>>> with open('MI_5MINS_HIST10505.csv', encoding='Big5') as f:
...     for row in csv.reader(f):
...         print(row)
... []
['2016年05月发行量加权股价指数历史数据']
['日期', '开盘指数', '最高指数', '最低指数', '收盘指数']
['105/05/03', '8,362.35', '8,363.59', '8,268.41', '8,294.12']
['105/05/04', '8,282.52', '8,285.96', '8,153.57', '8,185.47']
['105/05/05', '8,182.05', '8,207.98', '8,122.34', '8,167.96']
['105/05/06', '8,169.54', '8,170.68', '8,104.16', '8,146.43']
['105/05/09', '8,156.65', '8,177.36', '8,106.13', '8,131.83']
['105/05/10', '8,119.71', '8,168.13', '8,075.58', '8,156.29']
['105/05/11', '8,182.92', '8,204.51', '8,115.44', '8,135.56']
['105/05/12', '8,122.78', '8,162.44', '8,105.49', '8,108.05']
['105/05/13', '8,095.93', '8,095.93', '7,999.98', '8,053.69']
['105/05/16', '8,035.21', '8,091.48', '8,003.93', '8,067.60']
['105/05/17', '8,081.89', '8,151.20', '8,047.31', '8,140.48'] ['']
>>>
```

¹ 发行量加权股价指数历史数据: goo.gl/dk9hR

在这里使用的是 `csv` 的 `reader()` 来进行 CSV 的读取, `reader()` 实际上可以接受 `iterable` 对象 (可迭代的对象), 可对每次迭代返回的每一行 (row) 数据进行解析, 由于 `open()` 返回的文件对象就是 `iterable` 对象, 因此使用 `open()` 打开文件直接提供给 `reader()` 是常见的方式, `reader()` 返回的对象也是 `iterable`, 可直接使用 `for in` 语句进行迭代。

`reader()` 默认的 CSV 偏好格式是 'excel', `csv` 目前内建了 'unix'、'excel'、'excel-tab' 三种偏好格式, 可通过 `csv.list_dialects()` 来得知有哪些偏好格式, 在使用 `reader()` 时可使用 `dialect` 参数指定偏好格式。

除了指定 `dialect` 参数之外, 还有格式参数可以指定, 例如可以使用 `delimiter` 来指定分隔符, 使用 `quotechar` 指定引号字符。

```
csv.reader(csvfile, delimiter=',', quotechar='"')
```

可用的格式参数说明可参考 “Dialects and Formatting Parameters¹”。

如果想要输出 CSV 格式, 可将数据源组织为一个列表, 其中每个元素就是一行数据, 每行数据含有各个字段的信息。例如, 若想将先前下载的 CSV 文件转存为 UTF-8, 可以如下编写:

```
>>> with open('MI_5MINS_HIST10505.csv', encoding = 'GBK') as rf:
...     with open('10505-UTF8.csv', 'w', encoding = 'UTF-8', newline = '')
...     as wf:
...         rows = csv.reader(rf)
...         csv.writer(wf).writerows(rows)
...
>>>
```

要输出 CSV, 可以使用 `csv.writer()`, 实际上它可以接受任何具有 `write()` 方法的对象, 若使用文件对象, 记得 `newline` 要设为 "", 因为文件对象默认写出数据时是会换行的。 `csv.writer()` 同样也可以指定 `dialect` 参数以及一些格式参数。若想要逐行写出, 也可以使用 `writerow()` 方法。

● 使用 DictReader()、DictWriter()

除了将 CSV 以列表的方式进行处理之外, 也可以使用 `csv` 的 `DictReader()`、`DictWriter()` 将 CSV 以字典的方式处理。例如:

```
>>> custs = [
...     'first,last',
...     'Justin,Lin',
...     'Monica,Huang',
...     'Irene,Lin'
... ]
>>> for row in csv.DictReader(custs):
...     print(row)
...
{'last': 'Lin', 'first': 'Justin'}
{'last': 'Huang', 'first': 'Monica'}
{'last': 'Lin', 'first': 'Irene'}
>>>
```

`DictReader()` 实际上可以接受 `iterable` 对象, 可对每次迭代返回的每一行数据进行解析, 默认

¹ Dialects and Formatting Parameters: docs.python.org/3/library/csv.html#csv-fmt-params

会从第一行获取字段名，我们也可以使用 `fieldnames` 自行指定字段名。例如：

```
>>> custs = [
...     'Justin,Lin',
...     'Monica,Huang',
...     'Irene,Lin'
... ]
>>> for row in csv.DictReader(custs, fieldnames = ['firstname', 'lastname']):
...     print(row)
...
{'firstname': 'Justin', 'lastname': 'Lin'}
{'firstname': 'Monica', 'lastname': 'Huang'}
{'firstname': 'Irene', 'lastname': 'Lin'}
>>>
```

类似地，如果有一些字典想要写为 CSV，可以使用 `DictWriter()`，例如：

```
>>> custs = [
...     {'firstname': 'Justin', 'lastname': 'Lin'},
...     {'firstname': 'Monica', 'lastname': 'Huang'},
...     {'firstname': 'Irene', 'lastname': 'Lin'}
... ]
>>> with open('sample.csv', 'w', newline = '') as f:
...     writer = csv.DictWriter(f, fieldnames = ['firstname', 'lastname'])
...     writer.writeheader()
...     writer.writerows(custs)
...
>>>
```

同样地，若指定文件对象给 `DictWriter()`，记得必须将 `newline` 设为“”，`DictWriter` 有个 `fieldnames` 参数可以指定字段名，使用 `writeheader()` 可以写出字段名，使用 `writerows()` 可以将整个序列（每个元素是个字典）的内容写出，若打开 `sample.csv`，会有以下内容：

```
firstname,lastname
Justin,Lin
Monica,Huang
Irene,Lin
```

接下来的范例是个综合练习，可指定从证券交易所下载的发行量加权股价指数历史数据 CSV 文件以及想要查询的字段名，将查询结果逐行显示出来。



data_formats index_history.py

```
import csv  ← ❶ 导入 csv 模块
```

```
def csv_to_list(csvfile):
    with open(csvfile, encoding = 'Big5') as f:
        fieldnames = ['日期', '开盘指数', '最高指数', '最低指数', '收盘指数']
        reader = csv.DictReader(f, fieldnames = fieldnames)
        return list(reader)[2:-2]  ← ❷ 不需要第一行与最后一行的空白，以及第二行的字段名
```

```
def row_with_fields(row, fieldnames):  ← ❸ 每行只包含指定的字段
    return {fieldname : row[fieldname] for fieldname in fieldnames}
```

```
def index_with_fields(index_data, fieldnames):  ← ❹ 只收集指定的字段数据
    return (row_with_fields(origin_row, fieldnames) for origin_row in
            index_data)
```

```

csvfile = input('CSV 文件名: ')  ← ⑤ 将指定的字段按逗号分割为列表
fieldnames = input('查询字段: ').split(",")
index_data = csv_to_list(csvfile)

for name in fieldnames:  ← ⑥ 显示字段名
    print(name, end = '\t\t')
print()

for row in index_with_fields(index_data, fieldnames):  ← ⑦ 显示字段内容
    for name in fieldnames:
        print(row[name], end = '\t\t')
    print()

```

程序中首先导入了 `csv` 模块①，在 `csv_to_list()` 中使用 `DictReader()` 读取 CSV 文件，并且指定了字段名，从图 10-4 中可以看到，CSV 文件的第一行与最后一行是空白的，而第二行是字段名，这些我们都不需要，因此使用列表的切片操作将之去除②。

在 `row_with_fields()` 中会从指定的行中提取指定的字段③，在这里字典的 `for comprehension` 操作发挥了用途。在 `index_with_fields()` 中收集的数据将只有包含指定的字段④。

当程序启动时会让用户输入 CSV 文件名与想要显示的字段名，字段名称必须使用逗号分隔开，为了操作方便，使用 `split()` 按逗号分割成了列表⑤。程序中首先显示了字段名⑥，然后调用 `index_with_fields()` 提取指定的字段并显示出来⑦。

执行的结果如下所示：

CSV文件名: MI_5MINS_HIST10505.csv

查询字段: 日期,最高指数,收盘指数

日期	最高指数	收盘指数
2016/05/03	8,363.59	8,294.12
2016/05/04	8,285.96	8,185.47
2016/05/05	8,207.98	8,167.96
2016/05/06	8,170.68	8,146.43
2016/05/09	8,177.36	8,131.83
2016/05/10	8,168.13	8,156.29
2016/05/11	8,204.51	8,135.56
2016/05/12	8,162.44	8,108.05
2016/05/13	8,095.93	8,053.69
2016/05/16	8,091.48	8,067.60

10.3.2 JSON

JSON 全名是 JavaScript Object Notation (JavaScript 对象表示法)，它是 JavaScript 对象文字表示法 (Object Literal) 的子集，规范于 ECMA-404¹，也可以在“Introducing JSON²”找到详细的 JSON 格式说明，以及各种程序设计语言中可处理 JSON 的链接库。

一开始 JSON 是盛行于 JavaScript 生态圈的轻量交换格式，由于其易读、易写、易于解析且

¹ ECMA-404: www.ecma-international.org/publications/standards/Ecma-404.htm

² Introducing JSON: www.json.org

可提供层次结构，逐渐成了各种应用程序之间常用的交换格式之一。

整体而言，JSON 格式与 JavaScript 对象文字表示法（Literal）格式类似，有点巧合的是，Python 的语句可以极为相近地模仿 JavaScript 文字表示法，举个例子来说，下面的 JavaScript 程序代码片段使用对象文字表示法创建了一个对象。

```
var obj = {
  name : 'Justin',
  age : 40,
  childs : [
    {
      name : 'Irene',
      age : 8
    }
  ]
};
```

在 Python 中可以使用字典与列表等来模拟。

```
>>> obj = {
...     'name': 'Justin',
...     'age' : 40,
...     'childs': [
...         {
...             'name': 'Irene',
...             'age' : 8
...         }
...     ]
... }
>>>
```

这已经很接近 JSON 的对象格式了，还得注意的是在 JSON 的对象格式之中：

- 名称必须用双引号（"）包括。
- 值可以是双引号（"）包括的字符串，或者是数字、true、false、null、JavaScript 数组（相当于 Python 的列表）或子 JSON 格式。

举例来说，要将刚才的 obj 以 JSON 的对象格式表示，会如下所示：

```
jsonText = '{"name":"Justin","age":40,"childs":[{"name":"Irene","age":8}]}'
```

特意排版后会比较容易观察。

```
jsonText = '''{
"name" : "Justin",
"age" : 40,
"childs": [
    {
        "name" : "hamimi",
        "age" : 3
    }
]
}'''
```

实际上 JSON 不单只有对象格式，数字、true、false、null、使用""括起来的字符串等都是合法的 JSON 格式。

Python 内建了 json 模块, API 的使用类似 pickle, 将 Python 内建类型转为 JSON 格式的过程称为编码 (Encoding), 将 JSON 格式转为 Python 内建类型的过程称为解码 (Decoding), 在编码或解码时, Python 内建类型与 JSON 格式的对应关系如表 10.3 所示。

表 10.3 Python 内建类型与 JSON 的对应

Python	JSON
字典 (dict)	对象
列表 (list), 元组 (tuple)	数组
字符串 (str)	字符串
int, float	数字
True	true
False	true
None	null

使用 json.dumps()、json.dump()

如果要将 Python 内建类型编码为 JSON 格式, 那么可以使用 json.dumps()。例如:

```
>>> import json
>>> obj = {
...     'name': 'Justin',
...     'age': 40,
...     'childs': [{'name': 'Irene', 'age': 8}]
... }
>>> json.dumps(obj)
'{"name": "Justin", "childs": [{"name": "Irene", "age": 8}], "age": 40}'
>>>
```

json.dumps() 可用的参数很多, 这里介绍几个常用的, 像 sort_keys 可以指定为 True, 这会使得 JSON 格式输出时根据键进行排序, indent 参数可指定数字, 这会为 JSON 格式加上指定的空格数量进行缩排, 在显示 JSON 格式时会比较容易读。

```
>>> print(json.dumps(obj, sort_keys = True, indent = 4))
{
    "age": 40,
    "childs": [
        {
            "age": 8,
            "name": "Irene"
        }
    ],
    "name": "Justin"
}
>>>
```

实际上, 就算是简单地调用 json.dumps(obj)也做了些简单的易读性处理, 也就是逗号、冒号之后都有个空格, 这是因为 separators 默认是(, ', ':'), 如果指定为(, ', ':), 就不会有空格了, 像在进行网络传输时, 如果能省掉不必要的空格, 就可省去不必要的流量开销。

```
>>> json.dumps(obj)
'{"name": "Justin", "childs": [{"name": "Irene", "age": 8}], "age": 40}'
>>> json.dumps(obj, separators=(, ', ':))
'{"name": "Justin", "childs": [{"name": "Irene", "age": 8}], "age": 40}'
>>>
```

```
>>>
```

默认在将 Python 的字典编码为 JSON 对象格式时, 字典的键只能是字符串, 如果不是字符串, 就会引发 `ValueError` 错误, 如果将 `skipkeys` 指定为 `True`, 那么遇到非字符串的键就会略过。

刚才的示范都是针对 Python 内建类型, 如果调用 `json.dump()` 时指定了非内建类型, 默认会引发 `TypeError` 错误。

```
>>> class Customer:
...     def init (self, name, age):
...         self.name = name
...         self.age = age
...
>>> cust = Customer('Justin', 40)
>>> json.dumps(cust)
...略
TypeError: < main .Customer object at 0x010B0E50> is not JSON serializable
>>>
```

我们可以指定一个转换函数给 `default` 参数, 转换函数必须返回 Python 内建类型, 以进行 JSON 编码。例如:

```
>>> class Customer:
...     def init (self, name, age):
...         self.name = name
...         self.age = age
...     def json_serializable(self):
...         return {'name' : self.name, 'age' : self.age}
...
>>> json.dumps(cust, default = Customer.json_serializable)
{'age': 40, 'name': 'Justin'}
```

如果需要将编码后的 JSON 格式写到某个目标, 可以使用 `json.dump()`, 它的第二个参数接受一个具有 `write()` 方法的对象, 例如一个文件对象, 因此若要将对象编码为 JSON 并写至文件, 可以采用如下程序语句:

```
>>> with open('data.txt', 'w') as f:
...     json.dump(obj, f)
...
>>>
```

● 使用 `json.loads()`、`json.load()`

如果要将 JSON 格式解码为内建类型对象, 可以使用 `json.loads()`, 例如:

```
>>> jsonText = '{"name": "Justin", "age": 40, "childs": [{"name": "Irene", "age": 8}]}'
>>> json.loads(jsonText)
{'age': 40, 'childs': [{'age': 8, 'name': 'Irene'}], 'name': 'Justin'}
```

如果想将 JSON 格式解码为自定义类型实例, 可以在使用 `json.loads()` 时指定一个函数给 `object_hook`, 这个函数负责将内建类型转换为自定义类型实例。

```
>>> def to_cust(obj):
...     return Customer(obj['name'], obj['age'])
...
>>>
```



```
>>> cust = json.loads(jsonText, object_hook = to_cust)
>>> cust.name 'Justin'
>>> cust.age 40
>>>
```

如果想从某个来源加载 JSON 格式进行解码, 可以使用 `json.load()`, 它的第一个参数接受一个具有 `read()` 方法的对象, 例如文件对象, 因此若要从文件中读取 JSON 并解码, 如下编写:

```
>>> with open('test.txt') as f:
...     print(json.load(f))
...
{'name': 'Justin', 'childs': [{'name': 'Irene', 'age': 8}], 'age': 40}
>>>
```

10.3.3 XML

身为开发者, 对于 XML 必然不陌生, 它可用来表现具有层级结构的数据, 具有威力强大的描述能力, 简单的 XML 一目了然, 然而它也可以很复杂, 如果读者未曾听说过 XML, 或者想了解更多有关 XML 的说明, 建议读者参考“XML Tutorial¹”作为起点。

在处理 XML 时, Python 提供了几个模块, `xml.dom` 模块是基于 W3C DOM² (Document Object Model) 规范的实现, 最熟悉这套规范的应该是 JavaScript 开发者, DOM 需要将整个 XML 文件读入进行解析, 以便能够对文件的各个部分进行存取。

`xml.sax` 模块是基于 SAX (Simple API for XML) 的实现, SAX 并不存在一个标准, Java 对 SAX 的实现被视为一种非正式的规范, SAX 不会一次读入整个 XML 文件, 而是基于事件的 API, 一边读取 XML 文件一边进行解析, 开发者可针对解析过程感兴趣的各个事件进行处理, 因此适用于大型 XML 文件的处理。

然而实际上, 对于常见的 XML 处理, Python 建议使用 `xml.etree.ElementTree`。相对于 DOM 来说, `ElementTree` 更为简单而快速, 相对于 SAX 来说, 可以使用 `iterparse()` 在读取 XML 文件的过程中实时进行处理。

由于 XML 的处理是个范围很大的议题, 完整描述并不是这一节的目的, 因此接下来将只对 `xml.etree.ElementTree` 进行说明。

提示 >>>

`xml.etree.cElementTree` 模块是用 C 语言实现的模块, 接口上与 `xml.etree.ElementTree` 相同, 然而处理速度更快。不过并非每个平台上的 Python 都有 `cElementTree`, 读者可以使用以下方式尝试使用 `cElementTree`, 若没有则使用 `ElementTree`。

```
try:
import xml.etree.cElementTree as ET except ImportError:
import xml.etree.ElementTree as ET
```

¹ XML Tutorial: www.w3schools.com/xml/

² W3C DOM: www.w3.org/DOM/

解析 XML

接下来的 XML 解析将使用 Python 的 `xml.etree.ElementTree` 官方文档中简单的 XML 范例 `country_data.xml`。

```
<?xml version="1.0"?>
<data>
  <country name="Liechtenstein">
    <rank>1</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
  <country name="Singapore">
    <rank>4</rank>
    <year>2011</year>
    <gdppc>59900</gdppc>
    <neighbor name="Malaysia" direction="N"/>
  </country>
  <country name="Panama">
    <rank>68</rank>
    <year>2011</year>
    <gdppc>13600</gdppc>
    <neighbor name="Costa Rica" direction="W"/>
    <neighbor name="Colombia" direction="E"/>
  </country>
</data>
```

如果有一个 XML 文件，那么可以使用 `xml.etree.ElementTree` 的 `parse()` 来载入，它会返回 `ElementTree` 实例代表整个 XML 树，我们可以使用 `getroot()` 来获取根节点，这会返回一个 `Element` 实例，一个 `Element` 就代表 XML 中一个标签元素，它是 `iterable`（可迭代的），对其进行迭代，可以获取它的子元素。

例如，下面示范了如何获取 XML 中全部的标签名称。

```
>>> import xml.etree.ElementTree as ET
>>>
>>> def show_tags(elem, ident = ' '):
...     print(ident+elem.tag)
...     for child in elem:
...         show_tags(child, ident + ' ')
...
>>> tree=ET.parse('country_data.xml')
>>> show_tags(tree.getroot())
data
country
  rank
  year
  gdppc
  neighbor
  neighbor
country
  rank
  year
  gdppc
  neighbor
country
```

```
rank
year
gdppc
neighbor
neighbor
>>>
```

如果想获取标签间包含的文字，那么可以使用 `Element` 的 `text`，如果想获取标签上设置的属性，那么可以使用 `Element` 的 `attrib`，它会返回一个字典，键值分别就是属性名称与值。

注意 >>>

标签间的换行与缩排也会被视为标签文字，例如 `country_data.xml` 的 `<data>` 标签，如果使用其 `Element` 实例的 `text`，就会获取换行与缩排字符。

如果有一个 XML 字符串，那么可以使用 `fromstring()` 来解析 XML 字符串，它会直接返回一个 `Element` 实例，代表 XML 字符串的根节点。例如：

```
>>> import xml.etree.ElementTree as ET
>>>
>>> def show_tags(elem, ident = ' '):
...     print(ident+elem.tag)
...     for child in elem:
...         show_tags(child, ident + ' ')
...
>>> xml = '''<?xml version="1.0"?>
... <data>
...   <country name="Liechtenstein">
...     <rank>1</rank>
...     <year>2008</year>
...     <gdppc>141100</gdppc>
...     <neighbor name="Austria" direction="E"/>
...     <neighbor name="Switzerland" direction="W"/>
...   </country>
... </data>'''
>>> root = ET.fromstring(xml)
>>> root
<Element 'data' at 0x00E2AF30>
>>> root.tag 'data'
>>>
```

除了以迭代的方式来获取各个标签的 `Element` 实例之外，`ElementTree` 或 `Element` 还提供了 `find()`、`findall()`、`iterfind()` 等方法，可以指定 XPath 表达式 (XPath expressions¹) 来获取想要的标签。例如：

```
>>> tree = ET.parse('country_data.xml')
>>> tree.find('country')
<Element 'country' at 0x00F31420>
>>> tree.findall('country/neighbor')
[<Element 'neighbor' at 0x00F31F90>, <Element 'neighbor' at 0x00F31FC0>,
<Element 'neighbor' at 0x00F3E0F0>, <Element 'neighbor' at 0x00F3E1E0>,
<Element 'neighbor' at 0x00F3E210>]
```

¹ XPath expressions: www.w3.org/TR/xpath


```
>>> tree.iterfind('country/neighbor')
<generator object prepare_child.<locals>.select at 0x00F31360>
>>>
```

可以看到, `find()` 返回第一个找到的子元素, `findall()` 会以列表返回找到的全部子元素, `iterfind()` 会创建一个生成器, 可用来逐步迭代。有关可支持的 XPath 表达式可参考“XPath support¹”。

如果有一个 `Element` 想要直接获取 XML 字符串的 bytes 数据, 可以使用 `tostring()`。

```
>>> country = tree.find('country')
>>> ET.tostring(country)
b'<country name="Liechtenstein">\n      <rank>1</rank>\n
<year>2008</year>\n    <gdppc>141100</gdppc>\n    <neighbor direction="E"
name="Austria" />\n    <neighbor direction="W" name="Switzerland" />\n
</country>\n '
```

修改 XML

如果想要修改 XML 文件的内容, 可以使用 `Element` 的 `append()` 来附加一个元素, 使用 `insert()` 来插入元素, 使用 `remove()` 可以删除元素, 使用 `set()` 来设置元素属性等。例如, 可以为 `country_data.xml` 新增一个 `<country name="China"><rank>1</rank></country>`。

```
>>> country = ET.Element('country')
>>> country.set('name', 'China')
>>> rank = ET.Element('rank')
>>> rank.text = '1'
>>> country.append(rank)
>>> tree.getroot().append(country)
>>> print(ET.tostring(tree.getroot()).decode())
<data>
  <country name="Liechtenstein">
    <rank>1</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbordirection="E" name="Austria" />
    <neighbor direction="W" name="Switzerland" />
  </country>
  <country name="Singapore">
    <rank>4</rank>
    <year>2011</year>
    <gdppc>59900</gdppc>
    <neighbor direction="N" name="Malaysia" />
  </country>
  <country name="Panama">
    <rank>68</rank>
    <year>2011</year>
    <gdppc>13600</gdppc>
    <neighbor direction="W" name="Costa Rica" />
    <neighbor direction="E" name="Colombia" />
  </country>
<country name="China"><rank>1</rank></country></data>
>>> tree.write('sample.xml')
>>>
```

¹ XPath support: docs.python.org/3.5/library/xml.etree.elementtree.html#elementtree-xpath

从范例中可以看到,如果想将修改后的 `ElementTree` 写到 XML 文件中,可以使用它的 `write()` 方法。

● 渐进地解析 XML

如果打算一边读取 XML 一边进行解析,可以使用 `iterparse()`,可以在标签的'start'、'end'、'start-ns'、'end-ns'事件发生时进行相应的处理,默认 `iterparse()`只回报'end'事件,若要指定其他事件的回报,可以指定一个元组 (tuple) 给 `events` 参数。

例如,若想将 `country_data.xml` 的 `country` 标签中 `name` 属性取出,并置于`<country></country>`之间,可以如下编写程序:

```
>>> doc=ET.iterparse('country_data.xml', ('start', 'end'))
>>> for event, elem in doc:
...     if event == 'start' and elem.tag == 'country':
...         print('<country>{}'.format(elem.attrib['name']), end = '')
...     elif event == 'end' and elem.tag == 'country':
...         print('</country>')
...
<country>Liechtenstein</country>
<country>Singapore</country>
<country>Panama</country>
>>>
```

10.4 重点复习

如果要序列化 Python 对象,可以使用内建的 `pickle` 模块。使用 `pickle` 进行对象序列化时会将一个 Python 对象转换为字节流 (bytes),在 Python 的术语中,这个过程称为 `Pickling` (英文的原意为“腌制”),相反的操作则称之为 `unpickling`,会将字节流 (bytes) 转换为 Python 对象。

Python 使用 `pickling`、`unpickling` 来称呼,是为了避免与 `serialization` (串行化)、`marshalling` (编组) 等类似的名词混淆。

在 `pickle` 模块的使用上,如果想将对象转换为字节流 (bytes),那么可以使用 `dumps()` 函数,如果想将一个代表对象的字节流 (bytes) 转换为对象,那么可以使用 `loads()` 函数。

如果无法进行 `pickling` 或 `unpickling`,就会引发 `PicklingError` 错误或 `UnpicklingError` 错误 (父类都为 `PickleError`)。

`pickling` 时实际采用的模式是 Python 的专用格式, `pickle` 的保证是能向后兼容未来的新版本。

`shelve` 对象行为上像是字典的对象,键的部分必须是字符串,值的部分可以是 `pickle` 模块可处理的 Python 对象,它直接与一个文件关联,因此使用上就像一个简单的数据库接口。

DB-API 2.0 遵循 PEP 249 规范,所有的数据库接口都应该符合这个规范,以便编写程序时能有一致的方式,编写出来的程序也便于跨数据库执行。

Python 中内建了 SQLite 数据库,这是一个用 C 语言编写的轻量级数据库,数据库本身的数据可以存储在一个文件或者内存中,后者对于数据库应用程序的测试非常方便。

若想使用 SQLite 作为数据库,并编写 Python 程序与数据库进行操作,可以使用 `sqlite3` 模块,这个模块遵循 DB-API 2.0 的规范而实现。可以传给 `connect()` 一个 `':memory:'` 字符串,这样就

会在内存中创建一个数据库。

Connection 对象实现了上下文管理器，因此可以搭配 with 语句来使用，在 with 区块的操作完成之后，会自动 commit() 与 close()，若发生例外，则会自动 rollback()。

Cursor 本身是一个迭代器，每一次的迭代会调用 Cursor 的 fetchone() 方法。Cursor 的 execute() 方法本身可以将 SQL 语句参数化，有两种参数化的方式：使用问号 (?) 或具名占位符号。

交易的 4 个基本要求是原子性 (Atomicity)、一致性 (Consistency)、隔离行为 (Isolation behavior) 与持久性 (Durability)，按英文单词的首字母就简称为 ACID。

除了一些会隐含提交的情况之外，sqlite3 模块的默认实现并不会自动提交，因此我们必须自行调用 Connection 的 commit() 来进行提交，如果交易过程因为发生错误或其他情况，必须撤回交易时可以调用 Connection 的 rollback() 撤回操作。

不进行隔离，在多个连接同时存取数据库的情况下就会引发数据不一致的问题。

CSV 的全名为 Comma Separated Values (逗号分隔值)，是一种通用于电子表格、数据库之间的数据交换格式。Python 提供了 csv 模块，可隐藏 CSV 的读写细节，让开发人员轻松处理 CSV 格式。

JSON 全名 JavaScript Object Notation (JavaScript 对象表示法)，为 JavaScript 对象文字表示法的子集，规范于 ECMA-404，可以在 “Introducing JSON” 找到详细的 JSON 格式说明以及各种程序设计语言中可处理 JSON 的链接库。

在 JSON 的对象格式之中：

- 名称必须用双引号 (") 包括。
- 值可以是双引号 (") 括起来的字符串，或者是数字、true、false、null、JavaScript 数组 (相当于 Python 的列表) 或子 JSON 格式。

Python 内建了 json 模块，API 的使用类似 pickle。

对于常见的 XML 处理，Python 建议使用 xml.etree.ElementTree，相对于 DOM 来说，cElementTree 更为简单而快速，相对于 SAX 来说，可以使用 iterparse()，可以在读取 XML 文件的过程中实时进行处理。

课后练习

实践题

1. 请使用 sqlite3 模块改写 10.1.1 小节的 dvdlib_pickle.py，使之能将 DVD 信息保存到数据库中，假设 DVD 的名称是不重复的，并且有以下的执行结果：

```
dvd1 = DVD('Birds', 2016, 1, 'Justin Lin')
dvd1.save()
dvd2 = DVD.load('Birds')
print(dvd2) # 显示 DVD('Birds', 2016, 1, 'Justin Lin')
```

2. 请编写一个 dict_to_xml() 函数可以从一个简单的字典对象创建 XML 字符串，第一个参数可指定根标签。举例来说，若以 dict_to_xml('user', {'age': 40, 'name': 'Justin'}) 调用函数，它会返回 '<user><age>40 </age><name>Justin</name></user>'。

第 11 章

常用内建模块

学习目标

- 处理日期与时间
- 认识日志的使用
- 运用正则表达式
- 管理文件与目录

```
predicate, lt):  
    if len(greater) > num:  
        return len(greater) > num  
    return len(greater) > num  
C:\workspace\pdb_demo>python -m pdb filter_demo2.py  
-> def filter_it(predicate, lt):  
    if len(greater) > num:  
        return len(greater) > num  
    return len(greater) > num
```

```
1 -> def filter_it(predicate, lt):  
2     result = []  
3     for elem in lt:  
4         if predicate(elem):  
5             result.append(elem)  
6     return result  
7  
8 def len_greater_than(num):  
9     return len(greater) > num  
10  
11 return len(greater) > num  
(Fdb)
```



11.1 日期与时间

大多数开发者对于日期与时间通常都漫不经心，使用着似是而非的方式进行处理，因此在正式认识 Python 提供了哪些时间处理 API 之前，需要先来了解一些时间、日期的时空历史等议题，如此我们才会知道时间日期确实是个很复杂的议题，而使用程序来处理时间日期也不单只是使用 API 的问题。

11.1.1 时间的度量

想度量时间，要先有个时间基准，大多数人知道格林威治（Greenwich）时间，那么就先从这个时间基准开始认识。

▶ 格林威治标准时间

格林威治标准时间（Greenwich Mean Time, GMT）一开始是参考格林威治皇家天文台的标准太阳时间，格林威治标准时间的正午是太阳抵达天空最高点时。由于后面将讲到的一些缘由，GMT 时间常被不严谨（且有争议性）地当成 UTC 时间。

GMT 通过观察太阳而得，然而地球公转轨道为椭圆形且速度不一，本身自转也在缓慢减速中，因而会产生越来越大的时间误差，现在 GMT 已不作为标准时间使用。

▶ 世界时间

世界时间（Universal Time, UT）是通过观测远方星体跨过子午线（meridian）而得，这会比观察太阳准确一些。公元 1935 年，International Astronomical Union（国际天文学联合会）建议使用更精确的 UT 来取代 GMT，在 1972 年引入 UTC 之前，GMT 与 UT 是相同的。

▶ 国际原子时间

虽然观察远方星体会比观察太阳精确些，不过 UT 基本上仍受地球自转速度影响而有误差。1967 年定义的国际原子时间（International Atomic Time, IAT）将秒的国际单位制（International System of Units, SI）定义为铯（caesium）原子辐射振动 9192631770 周耗费的时间，时间从 UT 的 1958 年开始同步。

▶ 世界标准时间

由于基于铯原子振动定义的秒长是固定的，然而地球自转会越来越慢，因此实际上 IAT 时间会不断地超前基于地球自转的 UT 系列时间，为了保持 IAT 与 UT 时间差距不会过大，因而提出了具有折衷修正版本的世界标准时间（Coordinated Universal Time，或称为协调世界时间），常简称为 UTC。

UTC 经过了几次时间修正，为了简化日后对时间的修正，1972 年 UTC 采用了闰秒（leap second）修正（1 January 1972 00:00:00 UTC 实际上为 1 January 1972 00:00:10 IAT），确保 UTC 与 UT 相差不会超过 0.9 秒，加入闰秒的时间通常会在 6 月底或 12 月底由巴黎的 International Earth

Rotation and Reference Systems Service (国际地球自转与参考系统服务) 决定。

最近一次闰秒修正为 2012 年 6 月 30 日, 当时 IAT 实际上已超前 UTC 35 秒。

► Unix 时间

Unix 系统的时间表示法定义为 UTC 时间 1970 年 (Unix 元年) 1 月 1 日 00:00:00 为起点而经过的秒数, 不考虑闰秒修正, 用以表达时间轴上某一瞬间 (instant)。

► epoch (纪元)

某个特定时间的起点, 时间轴上某一瞬间。例如 Unix epoch 选为 UTC 时间 1970 年 1 月 1 日 00:00:00, 不少发源于 Unix 的系统、平台、软件等也都选择这个时间作为时间表示法的起算点, 例如稍后要介绍的 `time.time()` 返回的数字也是从 1970 年 (Unix 元年) 1 月 1 日 00:00:00 起经过的秒数。

提示 >>>

以上是关于时间日期的重要整理和说明, 足以了解后续 API 该如何使用, 如果有机会, 那么可以到维基百科上详细认识时间与日期: <http://zh.wikipedia.org/>

就以上这些说明来说几个重点:

- 就目前来说, 即使标注为 GMT (无论是文件说明还是 API 的日期时间字符串描述), 实际上谈到的时间指的是 UTC 时间。
- 秒的单位定义基于 IAT, 也就是铯原子辐射振动次数。
- UTC 考虑了地球自转越来越慢而有闰秒修正, 确保 UTC 与 UT 相差不会超过 0.9 秒。最近一次的闰秒修正为 2012 年 6 月 30 日, 当时 IAT 实际上已超前 UTC 35 秒。
- Unix 时间是 1970 年 1 月 1 日 00:00:00 为起点而经过的秒数, 不考虑闰秒, 不少发源于 Unix 的系统、平台、软件等也都选择这个时间作为时间表示法的起算点。

11.1.2 年历与时区简介

度量时间是一回事, 表达日期又是另一回事, 前面谈到的时间起点都是使用公历, 中文世界又常称为阳历或日历。在谈到公历之前得稍微谈一下其他历法。

► 儒略历

儒略历 (Julian calendar) 是现今公历的前身, 用来取代罗马历 (Roman calendar), 于公元前 46 年被 Julius Caesar 采纳, 公元前 45 年开始实施, 约于公元 4 年到 1582 年间广为各地采用。儒略历修正了罗马历隔三年设置一闰年的错误, 改采用四年一闰。

► 格里历

格里历 (Gregorian calendar) 改革了儒略历, 由教宗 Pope Gregory XIII 于 1582 年颁行, 将儒略历 1582 年 10 月 4 日星期四的后一天订为格里历 1582 年 10 月 15 日星期五。

不过由于各个国家改历的时间并不相同，像英国改历的时间是在 1752 年 9 月初，因此在 Unix/Linux 中查询 1752 年月历时会发现 9 月平白少了 11 天，如图 11-1 所示。

```

caterpillar@caterpillar-VirtualBox:~$ cal 1752
1752
  一月          二月          三月
日 一 二 三 四 五 六 日 一 二 三 四 五 六 日 一 二 三 四 五 六
 5 6 7 8 9 10 11  2 3 4 5 6 7 8  1 1 2 3 4 5 6 7
 5 6 7 8 9 10 11  2 3 4 5 6 7 8  8 9 10 11 12 13 14

  七月          八月          九月
日 一 二 三 四 五 六 日 一 二 三 四 五 六 日 一 二 三 四 五 六
 5 6 7 8 9 10 11  2 3 4 5 6 7 8  1 1 2 14 15 16
12 13 14 15 16 17 18  9 10 11 12 13 14 15 17 18 19 20 21 22 23
19 20 21 22 23 24 25 16 17 18 19 20 21 22 24 25 26 27 28 29 30
26 27 28 29 30 31 23 24 25 26 27 28 29
                        30 31
  
```

图 11-1 Linux 中查询 1752 年月历

ISO8601 标准

在一些相对来说较新的时间日期 API 应用场合中，我们可能会看到 ISO8601，严格来说 ISO8601 并非年历系统，它是时间日期表示方法的标准，用以统一时间日期的数据交换格式，像 yyyy-mm-ddTHH:MM:SS.SSS、yyyy-dddTHH:MM:SS.SSS、yyyy-Www-dTHH:MM:SS.SSS 之类的标准格式。

ISO8601 在数据定义上大部分与格里历相同，因而有些处理时间日期数据的程序或 API 为了符合时间日期数据交换格式的标准会采用 ISO8601。不过还是有些细微差别。像在 ISO8601 的定义中，19 世纪是指 1900~1999 年（包含该年），而格里历的 19 世纪是指 1801~1900 年（包含该年）。

时区

至于时区（Time zones），也许是各种时间日期的议题中最复杂的，每个地区的标准时间各不相同，因为这牵涉到地理、法律、经济、社会甚至政治问题。

从地理上来说，由于地球是圆的，基本上一边是白天另一边就是夜晚，为了让人们对时间的认知符合作息时间，因而设置了 UTC 偏移（offset），大致上来说，经度每 15 度是偏移一小时，考虑了 UTC 偏移的时间表示，通常会在时间的最后标上 Z 符号。

不过有些国家的领土横跨的经度很大，一个国家有多个时间反而造成困扰，因而不采取每 15 度偏移一小时的做法，像美国仅有 4 个时区，而中国、印度采用单一时区。

除了时区考虑之外，有些高纬度国家夏季、冬季日照时间差异很大，为了节省能源会尽量利用夏季日照，因而实施日光节约时间（Daylight saving time），也称为夏令时（Summer time）。基本上就是在实施的第一天，让白天的时间增加一小时，最后一天结束后再把一小时调整回来。中国也曾实施过夏令时，后来因为没太大实质作用而取消了，现在许多开发者多半不知道夏令时，偶尔会因此而踩到误区。举例来说，中国 1986 年 5 月 3 日 23 时 59 分 59 秒的下一秒从 1986 年 5 月 4 日 1 时 0 分 0 秒开始。

如果要认真面对时间和日期的处理，认识以上的基本信息是必要的，至少应该知道，一年的秒数绝对不是单纯的 $365 * 24 * 60 * 60$ ，更不应该基于这类错误的概念来进行时间与日期的计算。

11.1.3 使用 time 模块

如果想获取系统的时间，Python 的 time 模块提供了一个接口，用来调用各个平台上的 C 链接库函数，它提供的相关函数通常与 epoch 有关。

time()、gmtime()与 localtime()

尽管大多数平台都采取与 Unix 时间相同的 epoch(纪元)，也就是 1970 年 1 月 1 日 00:00:00 为起点。不过想确定自己平台上的 epoch，需调用 time.gmtime(0)来确认。

```
>>> import time
>>> time.gmtime(0)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=3,
tm_yday=1, tm_isdst=0)
>>>
```

gmtime()返回了 struct_time 实例，这是个具有 namedtuple 接口的对象，可以使用索引或属性名来获取对应的年、月、日等数值，采用的是 UTC（虽然 gmtime()名称上有 gm 字样），相关索引与属性名称会得到的值如表 11.1 所示。

表 11.1 struct_time 的索引与属性

索引	属性	值
0*	tm_year	年，例如 2016
1	tm_mon	月，范围 1~12
2	tm_mday	日，范围 1~31
3	tm_hour	时，范围 0~23
4	tm_min	分，范围 0~59
5	tm_sec	秒，范围 0~61
6	tm_wday	范围 0~6，星期一为 0
7	tm_yday	范围 1~366
8	tm_isdst	当前时区是否处于夏令时，1 为是，0 为否，-1 为未知

tm_sec 的值为 0~61，60 是为了闰秒，61 则是为了一些历史性的因素而存在。time.gmtime(0)表示从 epoch 算起经过了 0 秒，如果不指定数字，表示获取当前的时间并返回 struct_time 实例。获取当前的时间是指使用 time.time()获取从 epoch（使用 UTC）开始到当前经过的秒数。例如：

```
>>> time.gmtime()
time.struct_time(tm_year=2016, tm_mon=5, tm_mday=20, tm_hour=2, tm_min=30, tm_sec=33, tm_wday=4,
tm_yday=141, tm_isdst=0)
>>> time.gmtime(time.time())
time.struct_time(tm_year=2016, tm_mon=5, tm_mday=20, tm_hour=2, tm_min=30, tm_sec=34, tm_wday=4,
tm_yday=141, tm_isdst=0)
>>> time.time()
1463711442.941061
>>>
```

UTC 是一种绝对时间，与时区无关，也没有夏令时的问题，因此 `tm_isdst` 的值是 0。`time.time()` 返回的是浮点数，实际上获取的秒数的精度是不是能到秒以下的单位，要看系统而定。

简单来说，`time` 模块提供的是低级的机器时间，也就是从 `epoch` 起经过的秒数，然而有些辅助函数可以做些简单的转换，以便成为人类可理解的时间概念，除了 `gmtime()` 可获取 UTC 时间之外，`localtime()` 可提供当前所在时区的时间。同样地，`localtime()` 可指定从 `epoch` 起经过的秒数，不指定时则表示获取系统当前的时间，返回的是 `struct_time` 实例。

```
>>> time.localtime()
time.struct_time(tm_year=2016, tm_mon=5, tm_mday=20, tm_hour=10, tm_min=40, tm_sec=51, tm_wday=4,
tm_yday=141, tm_isdst=0)
>>> time.gmtime()
time.struct_time(tm_year=2016, tm_mon=5, tm_mday=20, tm_hour=2, tm_min=40, tm_sec=51, tm_wday=4,
tm_yday=141, tm_isdst=0)
>>>
```

看到了吗？中国的时区与 UTC 差了 8 小时，中国已经不实施夏令时了，因此 `tm_isdst` 的值是 0。

解析时间字符串

如果有个代表时间的字符串想要将其解析为 `struct_time` 实例，可以使用 `strptime()` 函数。例如：

```
>>> time.strptime('2016-05-26', '%Y-%m-%d')
time.struct_time(tm_year=2016, tm_mon=5, tm_mday=26, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=3,
tm_yday=147, tm_isdst=-1)
>>>
```

`strptime()` 的第一个参数指定代表时间的字符串，第二个参数为格式设置，可用的格式设置可参考 `time.strptime()`¹ 的说明。这是个从字符串解析而来的时间，未考虑时区信息，因此无法确定是否采用了夏令时，因此 `tm_isdst` 的值为 -1。

`gmtime()`、`localtime()` 与 `strptime()` 都可以返回 `struct_time` 实例。类似的，如果有个 `struct_time` 对象，想要转换为从 `epoch` 起经过的秒数，可以使用 `mktime()`。例如：

```
>>> d = time.strptime('2016-05-26', '%Y-%m-%d')
>>> time.mktime(d)
1464192000.0
>>>
```

时间字符串格式

可以使用 `ctime()` 获取一个简单的时间字符串描述，若不指定数字，则使用 `time()` 获取值。实际上，`ctime(secs)` 是 `asctime(localtime(secs))` 的封装，`asctime()` 可指定 `struct_time` 实例，获取一个简单的时间字符串描述，若不指定 `struct_time`，则使用 `localtime()` 返回值。

```
>>> time.ctime()
'Fri May 20 11:05:36 2016'
>>> time.asctime()
'Fri May 20 11:05:36 2016'
>>>
```

¹ `time.strptime()`: docs.python.org/3.5/library/time.html#time.strptime

当然，有时需要确定时间的字符串描述格式，这时可以使用 `strftime()`，它接受一个格式指定与 `struct_time` 实例。可用的格式设置同样可参考 `time.strftime()` 的说明，若不指定 `struct_time` 实例，则使用 `localtime()` 的值。例如，下面将 '2016-05-26' 转换为 '26-05-2016'。

```
>>> d = time.strptime('2016-05-26', '%Y-%m-%d')
>>> time.strftime('%d-%m-%Y', d)
'26-05-2016'
>>>
```

虽然有些辅助函数可以做些简单的转换，然而 `time` 模块提供的终究是低级的机器时间，用来表示人类可理解的时间概念并不方便，也不便于以人类可理解的时间单位来进行运算。若想从人类的观点来表示时间，可以进一步使用 `datetime` 模块。

11.1.4 使用 datetime 模块

人类在时间的表达上有时只需要日期，有时只需要时间，有时会同时表达日期与时间，通常不会特别声明时区，可能只会提及年、月、年月、月日、时分秒等。

🕒 datetime()、date()与 time()

对于人类的时间表达，`datetime` 模块提供了 `datetime`（包括日期与时间）、`date`（只有日期）、`time`（只有时间）等类来定义，下面是几个表示人类时间概念的范例。

```
>>> import datetime
>>> d = datetime.date(1986, 5, 26)
>>> (d.year, d.month, d.day)
(1986, 5, 26)
>>> t = datetime.time(11, 41, 35)
>>> (t.hour, t.minute, t.second, t.microsecond)
(11, 41, 35, 0)
>>> dt = datetime.datetime(1986, 5, 26, 11, 41, 35)
>>> (dt.year, dt.month, dt.day, dt.hour, dt.minute, dt.second)
(1986, 5, 26, 11, 41, 35)
>>>
```

提示 >>>

`datetime` 或 `date` 中的日期代表格里历，不过严格来说，是预期的格里历（Proleptic Gregorian calendar¹），因为它扩充了格里历，包含了 1582 年开始施行格里历前的日子。

以上的 `datetime`、`date`、`time` 默认是没有时区信息的，单纯用来表示一个日期或时间概念。`datetime()`、`date()`与 `time()`会进行基本的范围判断，如果设置了不存在的日期，例如 `date(2014, 2, 29)`，就会抛出 `ValueError` 例外，因为 2014 年并非闰年，不会有 2 月 29 日。

如果想使用今天的日期时间来创建 `datetime` 或 `date` 实例，可以使用 `datetime` 或 `date` 的 `today()`，如果想使用现在的时间来创建 `datetime` 实例，可以使用 `datetime` 的 `now()`。若想使用 UTC 时间来创建 `datetime` 实例，可以使用 `datetime` 的 `utcnow()`。例如：

¹ Proleptic Gregorian calendar: en.wikipedia.org/wiki/Proleptic_Gregorian_calendar

```
>>> from datetime import datetime, date
>>> datetime.today()
datetime.datetime(2016, 5, 23, 14, 55, 15, 762883)
>>> date.today()
datetime.date(2016, 5, 23)
>>> datetime.now()
datetime.datetime(2016, 5, 23, 14, 55, 24, 942699)
>>> datetime.utcnow()
datetime.datetime(2016, 5, 23, 6, 55, 30, 505080)
>>>
```

就算使用了 `datetime` 或 `date` 的 `today()`，或者是 `datetime` 的 `now()`、`utcnow()`，它们默认都是不带时区信息的，因此严格来说，我们不能说 `datetime.utcnow()` 创建的 `datetime` 实例代表着 UTC 时间，像上例最后的 `datetime(2016, 5, 23, 6, 55, 30, 505080)` 纯粹就只代表 2016 年 5 月 23 日 6 点 55 分 30 秒 505080 微秒这个时间概念罢了。

提示 >>>

尽管就 API 本身来说，`datetime`、`date`、`time` 本身不带时区信息，不过若程序运行时不需处理时区转换问题，通常所在时区就暗示是 `datetime`、`date`、`time` 的时区，因为人们若不特别提及时区，其实指本地时区居多。

如果有一个 `datetime` 或 `date` 实例，我们想将它们包含的时间概念转换为 UTC 时间戳 (Time stamp)，那么先通过 `datetime` 或 `date` 实例的 `timetuple()` 返回 `time.struct_time`，然后再通过 `time.mktime()` 将之转为 UTC 时间戳。例如：

```
>>> import time
>>> now = datetime.now()
>>> st = now.timetuple()
>>> time.mktime(st)
1463987245.0
>>>
```

如果有一个时间戳，也可以通过 `datetime` 或 `date` 的 `fromtimestamp()` 来创建 `datetime` 或 `date` 实例。例如：

```
>>> now = time.time()
>>> datetime.fromtimestamp(now)
datetime.datetime(2016, 5, 23, 15, 9, 36, 349939)
>>> date.fromtimestamp(now)
datetime.date(2016, 5, 23)
>>>
```

🕒 时间字符串描述与解析

`datetime`、`date`、`time` 实例都有个 `isoformat()` 方法，可以返回时间字符串描述，采用的是 ISO8601 标准，当日期与时间同时表示时，默认会使用 T 来分隔，如果必要也可以自行指定。例如：

```
>>> datetime.now().isoformat()
'2016-05-23T15:20:38.734850'
>>> datetime.now().isoformat(' ')
'2016-05-23 15:20:47.003517'
>>>
```

如果想要格式化时间字符串,那么 `datetime`、`date` 或 `time` 实例可以使用 `strftime()` 方法,使用方式类似于 `time` 模块的 `strftime()`,如果要解析时间字符串,那么可以使用 `datetime` 类的 `strptime()` 类方法,使用方式类似于 `time` 模块的 `strptime()`,只不过 `datetime` 类的 `strptime()` 类方法会返回 `datetime` 实例。例如:

```
>>> date.today().isoformat()
'2016-05-23'
>>> date.today().strftime('%d-%m-%Y')
'23-05-2016'
>>> datetime.strptime('2016-5-26', '%Y-%m-%d')
datetime.datetime(2016, 5, 26, 0, 0)
>>>
```

日期与时间的运算

如果需要进行日期或时间的运算,那么可以使用 `datetime` 类的 `timedelta` 类方法,可以创建的时间单位参数有 `days`、`seconds`、`microseconds`、`milliseconds`、`minutes`、`hours`、`weeks`,指定数字时可以是整数或浮点数。

例如,若有个 `datetime` 实例表示当前的时间,你想知道加上 3 周又 5 天 8 小时 35 分钟后的日期时间是什么,可以如下编写:

```
>>> from datetime import datetime, timedelta
>>> datetime.now() + timedelta(weeks = 3, days = 5, hours = 8, minutes = 35)
datetime.datetime(2016, 6, 19, 0, 19, 48, 842485)
>>>
```

2014 年 2 月 21 日加 9 天会是几月几号呢? 2014 年 3 月 2 日?

```
>>> date(2014, 2, 21) + timedelta(days = 9)
datetime.date(2014, 3, 1)
>>>
```

实际上是 2014 年 3 月 1 日,使用 `timedelta` 来进行日期与时间运算会比较可靠,原因在于它可以处理像闰年之类的问题。

考虑时区

对于日期与时间的处理议题,只要涉及时区,往往就会变得极端复杂,正如 11.1.2 小节所说的,这牵涉了地理、法律、经济、社会甚至政治问题。

`datetime` 实例本身默认并没有时区信息,此时只是单纯表示本地时间。然而,它也可以补上时区信息。`datetime` 类中的 `tzinfo` 类可以继承以实现时区的相关信息与操作。从 Python 3.2 开始,`datetime` 类新增了 `timezone` 类,它是 `tzinfo` 的子类,用来提供基本的 UTC 偏移时区的实现。

举个例子来说,在创建了一个 `datetime` 实例后,想要明确设置其表示 UT 时间,可以如下编写:

```
>>> from datetime import datetime, timezone
>>> utc = datetime(1986, 5, 26, 3, 20, 50, 0, tzinfo = timezone.utc)
>>> utc
datetime.datetime(1986, 5, 26, 3, 20, 50, tzinfo=datetime.timezone.utc)
>>>
```

现在,我们可以说上面的 `utc` 引用的实例代表 UTC 时间,如果想将 `utc` 转换为中国的时区,由于中国时区基本上就是偏移 8 小时,因此可以如下转换:


```
>>> tz = timezone(offset = timedelta(hours = 8), name = 'Asia/Beijing')
>>> asia_beijing = utc.astimezone(tz)
>>> asia_beijing
datetime.datetime(1986, 5, 26, 11, 20, 50, tzinfo=datetime.timezone(datetime.timedelta(0, 28800), 'Asia/Beijing'))
>>>
```

不过, Python 内建的 `timezone` 只单纯考虑了 UTC 偏移, 不考虑夏令时等其他因素, 如果需要 `timezone` 以外的其他时区定义, 那么可以额外安装 Python 社区贡献的 `pytz`¹ 模块, 它可以使用 `pip install pytz` 来安装, 一旦安装了 `pytz` 模块, 就可以轻松地创建一个带有中国时区的 `datetime` 了。

```
>>> import datetime, pytz
>>> datetime.datetime(1986, 5, 26, 3, 20, 50, 0, tzinfo = pytz.timezone('Asia/Beijing'))
datetime.datetime(1986, 5, 26, 3, 20, 50, tzinfo=<DstTzInfo 'Asia/Beijing' LMT+8:06:00 STD>)
>>>
```

`pytz` 的 `timezone` 也可以解决棘手的夏令时问题。例如在中国时区, 1986 年 5 月 4 日 0 时 30 分 0 秒这个时间其实是不存在的(注: 中国在 1986~1991 年采用了夏令时, 在 1986 年 5 月 4 日开始采用当年的夏令时), 因为当时实施了夏令时, 所以可以使用 `timezone` 的 `normalize()` 来修正时间:

```
>>> tz = pytz.timezone('Asia/Beijing')
>>> d = datetime.datetime(1986, 5, 4, 0, 30, 0, 0, tzinfo = tz)
>>> d
datetime.datetime(1986, 5, 4, 0, 30, tzinfo=<DstTzInfo 'Asia/Beijing' LMT+8:06:00 STD>)
>>> tz.normalize(d)
datetime.datetime(1986, 5, 4, 1, 24, tzinfo=<DstTzInfo 'Asia/Beijing' CDT+9:00:00 DST>)
>>>
```

如果在考虑时区之后想修正夏令时这类的时间问题, 谨记最后要对 `datetime` 实例使用 `normalize()` 方法, 就算使用 `timedelta` 进行时间运算也不例外。例如在中国时区, 1986 年 3 月 31 日 23 时 40 分 0 秒加一个小时的时间会是多少呢?

```
>>> d = datetime.datetime(1986, 5, 3, 23, 40, 0, 0, tzinfo = tz)
>>> d + datetime.timedelta(hours = 1)
datetime.datetime(1986, 5, 4, 0, 40, tzinfo=<DstTzInfo 'Asia/Beijing' LMT+8:06:00 STD>)
>>> tz.normalize(d + datetime.timedelta(hours = 1))
datetime.datetime(1986, 5, 4, 1, 40, tzinfo=<DstTzInfo 'Asia/Beijing' CDT+9:00:00 DST>)
>>>
```

单纯地对 `d` 加上 `datetime.timedelta(hours = 1)` 的结果是错的, 因为中国时区没有 1986 年 5 月 3 日 0 时 40 分这个时间, 使用了 `normalize()` 之后, 才能得到考虑了夏令时的正确时间。

若必须处理时区问题, 一个常见的建议是使用 UTC 来进行时间的存储或操作, 因为 UTC 是绝对时间, 不考虑夏令时等问题, 在必须使用当地时区的场合时再使用 `datetime` 实例的 `astimezone()` 进行转换。例如:

```
>>> utc_tz = datetime.timezone.utc
>>> utc = datetime.datetime(1986, 5, 3, 14, 59, 59, tzinfo = utc_tz)
>>> utc.astimezone(pytz.timezone('Asia/Beijing'))
datetime.datetime(1986, 5, 3, 22, 59, 59, tzinfo=<DstTzInfo 'Asia/Beijing' CST+8:00:00 STD>)
>>> utc2 = utc + datetime.timedelta(hours = 2)
```

¹ `pytz`: pypi.python.org/pypi/pytz

```
>>> utc2.astimezone(pytz.timezone('Asia/Beijing'))
datetime.datetime(1986, 5, 4, 1, 59, 59, tzinfo=<DstTzInfo 'Asia/Beijing' CDT+9:0
0:00 DST>)
>>>
```

可以看到, UTC 时间为 1986 年 5 月 3 日 14 时 59 分 59 秒时, 中国时区的时间是 1986 年 5 月 3 日 22 时 59 分 59 秒, 对 UTC 时间加两小时后, 因为在当时那个时段中国要开始切换到夏令时, 所以转为中国时区的正确时间是 1986 年 5 月 4 日 1 时 59 分 59 秒 (而不是 1986 年 5 月 4 日 0 时 59 分 59 秒)。

提示 >>>

如果需要像图 11-1 那样的 Unix 日历表示, 可以使用 `calendar` 模块¹。

11.2 日志

系统中有许多值得记录的信息, 例如例外发生之后, 有些例外值得显示给用户看, 对于开发人员或系统人员才有意义的例外, 则可以记录下来。那么该记录哪些信息 (时间、信息产生处……) 呢? 用什么方式记录 (文件、数据库、远程主机……) 呢? 记录格式 (控制台、纯文本、XML……) 是什么? 这些都是在记录时值得考虑的因素, 这些信息也不是简单使用 `print()` 就能解决的, 这时候就应该使用 Python 提供的 `logging` 模块来进行日志的工作。

11.2.1 简介 Logger

使用日志的起点是 `logging.Logger` 类, 一般来说, 一个模块只需要一个 `Logger` 实例, 因此尽管可以直接构建 `Logger` 实例, 不过建议通过 `logging.getLogger()` 来获取 `Logger` 实例。例如:

```
import logging
logger = logging.getLogger(__name__)
```

调用 `getLogger()` 时可以指定名称, 相同名称下获取的 `Logger` 会是同一个实例, 因此通常会使用 `__name__`, 因为在模块中 `__name__` 就是模块名称 (若模块在软件包中, 则会包含软件包名称)。

提示 >>>

如果直接使用 `python` 执行某个模块, 那么 `__name__` 的值会是 `'__main__'`。

实际上, 调用 `getLogger()` 时可以不指定名称, 这时会获取根 `Logger` (root logger), 这是什么意思? 这是因为 `Logger` 实例之间有父子关系, 可以使用点 (.) 来区分父子层级关系, 父层级相同的 `Logger` 父 `Logger` 的配置也相同。例如若有个 `Logger` 名称为 `'openhome'`, 则名称 `'openhome.some'` 与 `'openhome.other'` 的 `Logger` 的父 `Logger` 配置都是 `'openhome'` 名称的 `Logger` 配置。

因此, 如果使用了软件包来管理许多模块, 想要软件包中的模块在进行日志时都使用相同的

¹ `calendar` 模块: docs.python.org/3.5/library/calendar.html

父配置，可以在软件包的 `__init__.py` 文件中如下编写：

```
import logging
logger = logging.getLogger(__name__)
# 其他 logger 配置设置
```

在初次 `import`（导入）软件包中某个模块时，会先执行 `__init__.py` 中的程序，此时 `__name__` 会是软件包名称，例如如果软件包名称为 `openhome`，那么 `__name__` 就是 `'openhome'`，接着执行被 `import` 的模块时，例如 `some` 模块，模块中的 `__name__` 会是 `'openhome.some'`，如此就创建了 `Logger` 之间的父子层级关系。

获取 `Logger` 实例之后可以使用 `log()` 方法输出信息，输出信息时可以使用 `Level` 的静态成员指定信息层级（`Level`）。例如：

logging_demo basic_logger.py

```
import logging

logger = logging.getLogger(__name__)
logger.log(logging.DEBUG, 'DEBUG 信息')
logger.log(logging.INFO, 'INFO 信息')
logger.log(logging.WARNING, 'WARNING 信息')
logger.log(logging.ERROR, 'ERROR 信息')
logger.log(logging.CRITICAL, 'CRITICAL 信息')
```

执行结果如下：

```
WARNING 信息
ERROR 信息
CRITICAL 信息
```

怎么只看到 `logging.WARNING` 以下的信息？`logging` 中的 `DEBUG`、`INFO`、`WARNING`、`ERROR`、`CRITICAL` 代表着不同的日志等级，它们的实际值都是数字，分别为 10、20、30、40、50，还有个 `NOTSET` 的值是 0。

由于在上面的范例中还未曾对获取的 `Logger` 实例进行任何设置，因此使用根 `Logger` 的日志等级默认只有值大于 30，也就是 `WARNING`、`ERROR`、`CRITICAL` 的信息才会输出。

`Logger` 实例本身有个 `setLevel()` 可以使用，不过要记得 `Logger` 有层级关系，每个 `Logger` 处理完自己的日志操作后会再委托父 `Logger` 处理。就日志等级这部分来说，如果上面的 `logger` 使用 `setLevel()` 指定为 `logging.INFO`，那么调用 `logger.log(logging.INFO, 'INFO 信息')` 时，虽然可以通过实例本身的日志等级，但是继续委托给父 `Logger` 处理时，因为父 `Logger` 配置还是 `logging.WARNING`，结果信息还是不会被输出。

因此，在不调整父 `Logger` 配置的情况下直接设置 `Logger` 实例，设置为更严格的日志层级才会有实际的作用。例如：

logging_demo basic_logger2.py

```
import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.ERROR)
```



```

logger.log(logging.DEBUG, 'DEBUG 信息')
logger.log(logging.INFO, 'INFO 信息')
logger.log(logging.WARNING, 'WARNING 信息')
logger.log(logging.ERROR, 'ERROR 信息')
logger.log(logging.CRITICAL, 'CRITICAL 信息')

```

执行结果如下:

```

ERROR 信息
CRITICAL 信息

```

如果要调整根 **Logger** 的配置, 可以使用 `logging.basicConfig()`, 例如可以指定 `level` 参数来调整根 **Logger** 的日志等级。

logging_demo_basic_logger3.py

```

import logging

logging.basicConfig(level = logging.DEBUG)

logger = logging.getLogger(__name__)
logger.log(logging.DEBUG, 'DEBUG 信息')
logger.log(logging.INFO, 'INFO 信息')
logger.log(logging.WARNING, 'WARNING 信息')
logger.log(logging.ERROR, 'ERROR 信息')
logger.log(logging.CRITICAL, 'CRITICAL 信息')

```

执行结果如下:

```

DEBUG: __main__:DEBUG 信息
INFO: __main__:INFO 信息
WARNING: __main__:WARNING 信息
ERROR: __main__:ERROR 信息
CRITICAL: __main__:CRITICAL 信息

```

除了使用 **Logger** 的 `log()` 方法指定日志等级之外, 还可以使用 `debug()`、`info()`、`warning()`、`error()`、`critical()` 等便捷方法。除此之外, 对于例外, **Logger** 提供了 `exception()` 方法, 日志等级使用 **ERROR**, 不过用程序代码来实现语义会更加明确。

11.2.2 使用 Handler、Formatter 与 Filter

根 **Logger** 的日志信息默认会输出至 `sys.stderr` (也就是标准错误), 如果想要修改使之输出至文件, 可以使用 `logging.basicConfig()` 来指定 `filename` 参数。例如 `logging.basicConfig(filename = 'openhome.log')`, 如果子 **Logger** 实例没有设置自己的处理程序, 那么就都会输出至指定的文件。

子 **Logger** 实例可以通过 `addHandler()` 添加自己的处理程序, 举例来说, 若想要设置子 **Logger** 可以输出至文件, 可以如下编写程序:

logging_demo_handler_demo.py

```

import logging

logging.basicConfig(filename = 'openhome.log')

```

```
logger = logging.getLogger(__name__)
logger.addHandler(logging.FileHandler('errors.log'))

logger.log(logging.ERROR, 'ERROR 信息')
```

在这个范例中，设置了 `logging.basicConfig(filename = 'openhome.log')`，也在子 `Logger` 上添加了 `FileHandler`，要记得子 `Logger` 处理完日志信息之后还会委托给父 `Logger`，因此若父 `Logger` 也设置了处理程序，会使用设置的处理程序来处理日志信息，结果就是我们会看到 `openhome.log` 与 `errors.log` 两个文件。

在 `logging` 中提供了 `StreamHandler`、`FileHandler` 与 `NullHandler`，`StreamHandler` 可以指定输出至指定的串流，像 `stderr`、`stdout`。`FileHandler` 可以指定输出至文件，而 `NullHandler` 什么都不做，有时在开发链接库时并不是真的想输出日志，除了这三个基本的处理程序之外，更多高级的处理程序可以在 `logging.handlers` 模块¹中找到，像可与远程机器沟通的 `SocketHandler`、支持 `SMTP` 的 `SMTPHandler` 等。

提示 >>>

`logging.handlers` 模块中提供了大量的处理程序，如果这仍不能满足你的需求，那么可以继承 `logging.Handler` 或其他处理程序类再实现自己所需的处理程序，实现方式可参考 `logging.handlers` 模块中的源码。

处理程序在输出信息时，格式默认使用指定的信息，如果要自定义格式，那么可以通过 `logging.Formatter()` 创建 `Formatter` 实例，再通过处理程序的 `setFormatter()` 设置 `Formatter` 实例。例如显示信息时若想连带显示时间、`Logger` 名称、日志等级，可以如下编写：

logging_demo formatter_demo.py

```
import logging, sys

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s')
handler = logging.StreamHandler(sys.stderr)
handler.setFormatter(formatter)

logger = logging.getLogger(__name__)
logger.addHandler(handler)

logger.log(logging.ERROR, '发生了 XD 错误')
```

`%(asctime)` 使用人类可理解的时间格式来显示日志的时间，`%(name)` 显示 `Logger` 名称，`%(levelname)` 显示日志层级，`%(message)` 显示指定的日志信息，除了这些格式设置之外，还有其他许多可用的设置，这部分是由 `LogRecord` 的属性来定义，可直接参考官方的在线文档²。

上面的范例执行结果如下所示：

¹ `logging.handlers` 模块: docs.python.org/3.5/library/logging.handlers.html

² `LogRecord` attributes: docs.python.org/3.5/library/logging.html#logrecord-attributes

```
2016-05-25 15:05:20,385 - __main__ - ERROR - 发生了 XD 错误
```

如果格式指定中出现了%(asctime)，内部会调用 `formatTime()` 来进行时间的格式化，如果想控制时间的格式，可以在使用 `logging.Formatter()` 时指定 `datefmt` 参数，指定的格式会使用 `time.strftime()` 来进行格式化。

如果不喜欢%这个字符，可以在使用 `logging.Formatter()` 时用 `style` 参数指定其他字符。

除了使用 `DEBUG`、`INFO`、`WARNING`、`ERROR`、`CRITICAL` 等日志等级过滤信息是否输出之外，如果还想要使用其他条件来过滤哪些信息可以输出，那么可以定义过滤器。可以通过继承 `logging.Filter` 类并定义 `filter(record)` 方法或者定义一个对象具有 `filter(record)` 方法根据传入的 `LogRecord` 获取日志时的信息，并返回 0 决定输出信息，返回非 0 值决定不输出信息。

不过，自 Python 3.2 之后，也可以使用函数作为过滤器，`Logger` 或 `Handler` 实例都有 `addFilter()` 方法，可以新增过滤器。下面是个简单的范例：

logging_demo filter_demo.py

```
import logging, sys

logger = logging.getLogger(__name__)
logger.addFilter(lambda record: 'Orz' in record.msg)

logger.log(logging.ERROR, '发生了 XD 错误')
logger.log(logging.ERROR, '发生了 Orz 错误')
```

在这个范例中，针对某个字词进行了过滤，只有信息中包含'Orz'才会显示出来。

```
发生了 Orz 错误
```

有关于 `LogRecord` 可用的属性同样可参考官方的在线文档。

提示 >>>

设置过滤器时别忘了，子 `Logger` 实例过滤后的信息还会再委托父 `Logger`，因此，若无法通过父 `Logger` 的过滤，信息仍旧不会显示出来。关于 `Logger` 进行日志的整个流程可参考“Logging Flow¹”。

11.2.3 使用 logging.config

以上都是使用程序编写方式来改变 `Logger` 对象的配置。实际上，可以通过 `logging.config` 模块使用配置文件来设置 `Logger` 配置。这也很方便，例如程序开发阶段，在配置文件中设置 `WARNING` 等级的信息就可输出，在程序上线之后，若想关闭不会影响程序运行的警告信息日志，以减少程序不必要的输出（不必要的日志输出会影响程序运行的效率），则只要在配置文件中做个修改即可。

不过配置文件的设置细节非常多而且复杂，为了让大家有个起点，在 Python 的 `logging` 模块的官方文档中有一个范例，是设置 `Logger` 配置的参考范例，当中列举的是一个使用程序配置的例子，

¹ Logging Flow: docs.python.org/3.5/howto/logging.html#logging-flow

如下所示:

```
import logging

# 创建 logger
logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)

# 创建控制台处理程序并设置日志等级为 DEBUG
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

# 创建 formatter
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# 将 formatter 设给 ch
ch.setFormatter(formatter)

# 将 ch 加入 logger
logger.addHandler(ch)

# 应用程序的程序代码
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

这个程序被简化了, 相关配置信息使用 `logging.config.fileConfig()` 从配置文件读取, 不过那是旧式的方法, 配置文件的编写格式是 ini, 不方便也不易于阅读, 有兴趣可以自行参考。

从 Python 3.2 开始, 建议改用 `logging.config.dictConfig()`, 可使用一个字典对象来设置配置信息, 因此这里改写官方的例子, 改用 `logging.config.dictConfig()`, 刚才程序配置的内容被改写如下:

logging_demo config_demo.py

```
import logging, logconf
import logging.config

logging.config.dictConfig(logconf.LOGGING_CONFIG)

# 创建 logger
logger = logging.getLogger('simple_example')

# 应用程序的程序代码
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

其中 `logconfig` 模块就是一个编写了配置信息的 `logconfig.py` 文件。

logging_demo logconf.py

```
LOGGING_CONFIG = {
    'version': 1,
    'handlers': {
```

```

    'console': {
        'class': 'logging.StreamHandler',
        'level': 'DEBUG',
        'formatter': 'simpleFormatter'
    }
},
'formatters': {
    'simpleFormatter': {
        'format': '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    }
},
'loggers': {
    'simple_example': {
        'level': 'DEBUG',
        'handlers': ['console']
    }
}
}

```

作为配置文件的 `logconf.py` 本身就是 Python 源码，进行配置设置时使用的是字典 (dict)，最重要的是必须有个 `'version'` 键名称，每个处理程序、格式器、过滤器或 `Logger` 实例都要有个名称，以便设置时引用。有关字典中可使用的键名称可参考“Dictionary Schema Details¹”的说明。

若不想使用 `.py` 作为配置文件，则可以使用 JSON，例如创建一个 `logconf.json` 文件。

logging_demo logconf.json

```

{
  "version": 1,
  "handlers": {
    "console": {
      "class": "logging.StreamHandler",
      "level": "DEBUG",
      "formatter": "simpleFormatter"
    }
  },
  "formatters": {
    "simpleFormatter": {
      "format": "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
    }
  },
  "loggers": {
    "simple_example": {
      "level": "DEBUG",
      "handlers": ["console"]
    }
  }
}

```

要留意的是，在 JSON 的规范中，必须使用双引号来括住键名称。有了这个 JSON 文件，就可以运用 10.3.2 小节介绍的 `json.load()` 来读取 JSON，并作为 `logging.config.dictConfig()` 的自变量。

¹ Dictionary Schema Details: docs.python.org/3.5/library/logging.config.html#dictionary-schema-details

logging_demo config_demo2.py

```
import logging, json
import logging.config

with open('logconf.json') as config:
    LOGGING_CONFIG = json.load(config)
    logging.config.dictConfig(LOGGING_CONFIG)

# 创建 logger
logger = logging.getLogger('simple_example')

# 应用程序的程序代码
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

11.3 正则表达式

正则表达式 (Regular expression) 最早是由数学家 Stephen Kleene 于 1956 年提出来的, 主要用于字符以及字符串格式对比, 后来在信息领域广为应用。Python 提供了一些支持正则表达式操作的标准 API, 下面将从如何定义正则表达式开始介绍。

11.3.1 正则表达式简介

如果我们有个字符串, 想根据某个字符或字符串进行分割, 可以使用 `str` 的 `split()` 方法, 它会返回分割后各子字符串组成的列表。例如:

```
>>> 'Justin,Monica,Irene'.split(',')
['Justin', 'Monica', 'Irene']
>>> 'JustinOrzMonicaOrzIrene'.split('Orz')
['Justin', 'Monica', 'Irene']
>>> 'Justin\tMonica\tIrene'.split('\t')
['Justin', 'Monica', 'Irene']
>>>
```

如果分割字符串的依据不单只是某个字符或子字符串, 而是任意单一数字呢? 例如, 想要将 'Justin1Monica2Irene' 按数字分割呢? 这个时候 `str` 的 `split()` 派不上用场, 需要的是正则表达式。在 Python 中, 使用 `re` 模块来支持正则表达式。如果想分割字符串, 那么可以使用 `re.split()` 函数。

```
>>> import re
>>> re.split(r'\d', 'Justin1Monica2Irene')
['Justin', 'Monica', 'Irene']
>>> re.split(r',', 'Justin,Monica,Irene')
['Justin', 'Monica', 'Irene']
>>> re.split(r'Orz', 'JustinOrzMonicaOrzIrene')
['Justin', 'Monica', 'Irene']
>>> re.split(r'\t', 'Justin\tMonica\tIrene')
['Justin', 'Monica', 'Irene']
>>>
```


在这里使用了 `re` 模块的 `split()` 函数，第一个参数要以字符串来指定正则表达式，在正则表达式中，`\d` 表示符合一个数字。

实际上若使用 Python 的字符串表示时，因为 `\` 在 Python 字符串中被作为转义 (Escape) 字符，因此要编写正则表达式时必须编写为 `"\\d"`。这样当然很麻烦，幸好在 Python 中可以在字符串前加上 `r`，表示这是个原始字符串 (Raw string)，就不用对 `\` 做任何转义操作，因此在编写正则表达式时，建议使用原始字符串。

Python 支持大多数标准的正则表达式，在使用 `re` 模块之前，认识正则表达式是必要的。正则表达式基本上包括两种字符：字面字符 (Literals) 与诠释字符 (Metacharacters)。

字面字符是指按照字面意义对比的字符，像刚才在范例中指定的 `Orz`，指的是三个字面字符 `O`、`r`、`z` 的规则；诠释字符是不按照字面对比，在不同场景有不同意义的字符，例如 `^` 是诠释字符，正则表达式 `^Orz` 是指行首立即出现 `Orz` 的情况，也就是此时 `^` 表示一行的开头，但正则表达式 `[^Orz]` 是指不包括 `O` 或 `r` 或 `z` 的对比，也就是在 `[]` 中时 `^` 表示不是之后几个字符的情况。诠释字符就像程序设计语言中的控制结构之类的语法，找出并理解诠释字符想要诠释的概念，对于正则表达式的阅读非常重要。

❶ 字符表示

字母和数字在正则表达式中都是按照字面意义对比的，有些字符之前加上了 `\` 之后，会被当作诠释字符，例如 `\t` 代表按【Tab】键的字符，表 11.2 列出了正则表达式支持的字符表示。

表 11.2 字符表示

字符	说明
字母或数字	对比字母或数字
<code>\\</code>	对比 <code>\</code> 字符
<code>\num</code>	<code>num</code> 用来指定字符的八进制编码
<code>\xnum</code>	<code>num</code> 用来指定字符的十六进制编码
<code>\uhhhh</code>	十六进制 <code>0xhhhh</code> 字符
<code>\Uhhhhhhhh</code>	十六进制 <code>0xhhhhhhhh</code> 字符
<code>\n</code>	换行 (<code>\u000A</code>)
<code>\v</code>	垂直定位 (<code>\u000B</code>)
<code>\f</code>	换页 (<code>\u000C</code>)
<code>\r</code>	回车 (<code>\u000D</code>)
<code>\a</code>	响铃 (<code>\u0007</code>)
<code>\b</code>	退格 (<code>\u0008</code>)
<code>\t</code>	Tab (<code>\u0009</code>)

诠释字符在正则表达式中有特殊意义，例如 `!$^*()+={}[]|\.:.?等`，若要对比这些字符，则必须加上忽略符号，例如若要对比 `!`，则必须使用 `\!`，若要对比 `$` 字符，则必须使用 `\$`。如果不确定哪些标点符号字符要加上忽略符号，可以在每个标点符号前加上 `\`，例如对比逗号也可以写成 `\,`。

如果正则表达式为 `XY`，那么表示对比“`X` 之后要紧跟着 `Y`”，如果想表示“`X` 或 `Y`”，可以使

用 $X|Y$ ，如果有多个字符要以“或”的方式表示，例如“X 或 Y 或 Z”，则可以使用稍后会介绍的字符类表示为 $[XYZ]$ 。

● 字符类

在正则表达式中，多个字符可以归类在一起，成为一个字符类（**Character class**），字符类会对比文字中是否有“任一个”字符符合字符类中某个字符。正则表达式中被放在 $[]$ 中的字符就成为一个字符类。例如，若字符串为 'Justin1Monica2Irene3Bush'，想要按 1 或 2 或 3 分割字符串，正则表达式可编写为 $[123]$ ：

```
>>> re.split(r'[123]', 'Justin1Monica2Irene3Bush')
['Justin', 'Monica', 'Irene', 'Bush']
>>>
```

正则表达式 123 连续出现字符 1、2、3，然而 $[]$ 中的字符是“或”的概念，也就是 $[123]$ 表示“1 或 2 或 3”， $|$ 在字符类只是个普通字符，不会被当作“或”。

字符类中可以使用连字符（-）作为字符类诠释字符，表示一段文字范围，例如要对比文字中是否有 1~5 任一数字出现，正则表达式为 $[1-5]$ ；要对比文字中是否有 a~z 任一字母出现，正则表达式为 $[a-z]$ ；要对比文字中是否有 1~5、a~z、M~W 任一字符出现，正则表达式可以写为 $[1-5a-zA-M-W]$ 。字符类中可以使用 $^$ 作为字符类诠释字符， $^$ 表示“非”（即“不是”）字符类（**Negated character class**），例如 $^[abc]$ 表示对比 a、b、c 以外的其他字符。表 11.3 为字符类范例列表。

表 11.3 字符类

字符类	说明
$[abc]$	a 或 b 或 c 任一字符
$^[abc]$	a、b、c 以外的任一字符
$[a-zA-Z]$	a~z 或 A~Z 任一字符
$[a-d[m-p]]$	a~d 或 m~p 任一字符（并集），等于 $[a-dm-p]$
$[a-z&[def]]$	a~z 且是 d、e、f 的任一字符（交集），等于 $[def]$
$[a-z&^[bc]]$	a~z 且不是 b 或 c 的任一字符（减集），等于 $[ad-z]$
$[a-z&^[m-p]]$	a~z 且不是 m~p 的任一字符，等于 $[a-lq-z]$

可以看到，字符类中可以再有字符类，把正则表达式想成是语言，字符类就像其中独立的子语言。

有些字符类很常用，例如经常对比是否为 0~9 的数字，可以编写为 $[0-9]$ ，或是编写为 \d ，这类字符被称为字符类缩写或预定义字符类（**Predefined character class**），它们不用被包括在 $[]$ 中，表 11.4 列出了可用的预定义字符类。

表 11.4 预定义字符类

预定义字符类	说明
.	任一字符
\d	对比任一数字字符，即 $[0-9]$
\D	对比任一非数字字符，即 $^[0-9]$
\s	对比任一空格符，即 $[\t\n\r\x0B\f]$

(续表)

预定义字符类	说明
\S	对比任一非空格符，即 <code>[^\s]</code>
\w	对比任一 ASCII 字符，即 <code>[a-zA-Z0-9_]</code>
\W	对比任一非 ASCII 字符，即 <code>[^\w]</code>

● 贪婪、惰性量词

如果想对比用户输入的手机号码格式是否为 XXXX-XXXXXXX，其中 X 为数字，虽然正则表达式可以使用 `\d\d\d\d-\d\d\d\d\d\d`，不过更简单的写法是 `\d{4}-\d{6}`，`{n}` 是贪婪量词（Greedy quantifier）表示法的一种，表示前面的项出现 n 次。表 11.5 列出了可用的贪婪量词。

表 11.5 贪婪量词

贪婪量词	说明
X?	X 项出现一次或没有
X*	X 项出现零次或多次
X+	X 项出现一次或多次
X{n}	X 项出现 n 次
X{n,}	X 项至少出现 n 次
X{n,m}	X 项出现 n 次但不超过 m 次

贪婪量词之所以贪婪，是因为看到贪婪量词时，对比器（Matcher，或称为匹配器）会把剩余文字全部吃掉，再逐步吐出（back-off）文字，看看是否符合贪婪量词后的正则表达式，如果吐出部分符合，吃下部分也符合贪婪量词就对比成功，结果就是贪婪量词会尽可能地找出长度最长的符合要求的文字。

例如文字 `xfoooooxxxfoo`，若使用正则表达式 `*foo` 对比，对比器会先吃掉整个 `xfoooooxxxfoo`，再吐出符合 `foo` 的部分，剩下的 `xfoooooxxx` 也符合 `*` 部分，所以得到的合乎要求的字符串就是整个 `xfoooooxxxfoo`。

如果在贪婪量词表示法后加上 `?`，将会成为惰性量词（Reluctant quantifier），又称为不情愿的量词，或非贪婪（non-greedy）量词（相对于贪婪量词来说），对比器看到惰性量词时，会一边吃掉剩余文字，一边看吃下的文字是否符合正则表达式，结果就是惰性量词会尽可能地找出长度最短的符合要求的文字。

例如文字 `xfoooooxxxfoo` 若用正则表达式 `*?foo` 对比，对比器在吃掉 `xfoo` 后发现符合 `*?foo`，接着继续吃掉 `xxxxxxfoo` 发现符合，所以得到 `xfoo` 与 `xxxxxxfoo` 两个符合要求的文字。

我们可以使用 `re` 模块的 `findall()` 来看两个量词的差别。

```
>>> re.findall(r'.*foo', 'xfoooooxxxfoo')
['xfoooooxxxfoo']
>>> re.findall(r'.*?foo', 'xfoooooxxxfoo')
['xfoo', 'xxxxxxfoo']
>>>
```

● 边界对比

如果有段文字 `Justin dog Monica doggie Irene`，若想直接按照其中的单词 `dog` 切出前后两个子字

字符串，也就是 Justin 与 Monica doggie Irene 两个部分，那么结果会让我们失望。

```
>>> re.split(r'dog', 'Justin dog Monica doggie Irene')
['Justin ', ' Monica ', 'gie Irene']
>>>
```

在程序中，doggie 因为当中有 dog 子字符串，也被当作分割的依据，才会有以上的结果，我们可以使用\b 标出单词边界，例如\bdog\b，这就只会对比出 dog 单词。例如：

```
>>> re.split(r'\bdog\b', 'Justin dog Monica doggie Irene')
['Justin ', ' Monica doggie Irene']
>>>
```

由于边界对比用来表示文字必须符合指定的边界条件，也就是定位点，因此这类表达式也常称为锚点（Anchor），表 11.6 列出了正则表达式中可用的边界对比。

表 11.6 边界对比

边界对比	说明
^	一行开头
\$	一行结尾
\b	单词边界
\B	非单词边界
\A	输入开头
\G	前一个符合项的结尾
\Z	非最后终端（final terminator）的输入结尾
\z	输入结尾

● 分组与引用

可以使用()来将正则表达式分组，除了作为正则表达式之外，还可以搭配量词使用。例如想要验证电子邮件格式，允许的用户名称开头要是大小写英文字母，之后可搭配数字，正则表达式可以写为^[a-zA-Z]+\d*，因为@后字段名可以有数层，必须是大小写英文字母或数字，正则表达式可以写为([a-zA-Z0-9]+\.)+，其中使用()给正则表达式分组，之后的+表示这个分组的表达式符合一次或多次，最后要以 com 结尾，整个结合起来的正则表达式就是^[a-zA-Z]+\d*@([a-zA-Z0-9]+\.)+com。

被分组的正则表达式还可以在稍后回头引用（Back reference），在这之前我们必须知道分组计数，如果有个正则表达式((A)(B(C)))中有 4 个分组，以遇到的左括号来计数，所以 4 个分组分别是：((A)(B(C)))、(A)、(B(C))以及(C)。

分组回头引用时，是在\b后加上分组计数，表示引用第几个分组的对比结果。例如\d\d 要求对比两个数字 (\d\d)\1，表示要输入 4 个数字，输入的前两个数字与后两个数字必须相同。例如输入 1212 会符合，因为 12 符合(\d\d)，\1 要求接下来的输入也要是 12；若输入 1234 则不符合，因为 12 符合(\d\d)，\1 要求接下来的输入也要是 12，然而接下来的数字是 34，所以不符合。

再来看个实用的例子，[""]^[^"]*"对比单引号或双引号中 0 或多个字符，但没有对比两个都要是单引号或双引号，([""]^[^"]*"则对比出前后引号必须一致。

提示>>>

想要得到有关正则表达式更完整的说明，除了可以参考 `re` 模块的文件说明，也可以参考“Regular Expression HOWTO¹”。

11.3.2 Pattern 与 Match 对象

在程序中使用正则表达式必须先对正则表达式进行解析、验证等操作，确定正则表达式语法无误，才能对字符串进行对比，如果不是频繁性地使用正则表达式，可以直接使用 `re` 模块的 `split()`、`findall()`、`search()`、`sub()`、`match()` 等函数。

创建 Pattern 对象

不过，解析、验证正则表达式往往是最耗时间的阶段，在频繁使用某正则表达式的场合，若可以将解析、验证过后的正则表达式重复使用，将提高效率。

`re.compile()` 可以创建正则表达式对象，在解析、验证过正则表达式无误后，返回的正则表达式对象可以重复使用。例如：

```
regex = re.compile(r'.*foo')
```

`re.compile()` 函数可以指定 `flags` 参数，进一步设置正则表达式对象的行为，例如不想分大小写对比 `dog` 文字，可以如下编写：

```
regex = re.compile(r'dog', re.IGNORECASE)
```

也可以在正则表达式中使用嵌入标志表示法（**Embedded Flag Expression**）。例如与 `re.IGNORECASE` 等效的嵌入标志表示法为 `(?i)`，下面的程序片段效果等同上例。

```
regex = re.compile('(?i)dog')
```

在获取正则表达式对象后，可以使用 `split()` 方法指定字符串按正则表达式进行分割，效果等同于使用 `re.split()` 函数；`findall()` 方法找出符合的全部子字符串，效果等同于使用 `re.findall()` 函数。

```
>>> dog = re.compile('(?i)dog')
>>> dog.split('The Dog is mine and that dog is yours')
['The ', ' is mine and that ', ' is yours']
>>> dog.findall('The Dog is mine and that dog is yours')
['Dog', 'dog']
>>>
```

使用 Match 对象

如果想获取对比匹配时更进一步的信息，可以使用 `finditer()` 方法，它会返回一个 `iterable` 对象，每一次迭代都会得到一个 `Match`（匹配）对象，可以使用它的 `group()` 来获取符合整个正则表达式的子字符串，使用 `start()` 来获取子字符串的起始索引，`end()` 来获取结尾索引。例如：

```
>>> dog = re.compile('(?i)dog')
>>> for m in dog.finditer('The Dog is mine and that dog is yours'):
```

¹ Regular Expression HOWTO: docs.python.org/3.5/howto/regex.html

```
... print(m.group(), 'between', m.start(), 'and', m.end())
...
Dog between 4 and 7
dog between 25 and 28
>>>
```

`search()`方法与 `match()`方法必须小心区分, `search()`会在整个字符串中寻找第一个符合的子字符串, 而 `match()`只会在字符串开头看接下来的字符串是否匹配, `search()`方法与 `match()`若符合, 都会返回 `Match` 对象, 否则返回 `None`。

```
>>> dog.search('The Dog is mine and that dog is yours')
<_sre.SRE_Match object; span=(4, 7), match='Dog'>
>>> dog.match('The Dog is mine and that dog is yours')
>>> dog.match('Dog is mine and that dog is yours')
<_sre.SRE_Match object; span=(0, 3), match='Dog'>
>>>
```

分组处理

如果正则表达式中设置了分组, `findall()`方法会以列表返回各个分组。如前文所述, 使用 `(\d\d)\1` 时, 表示要输入 4 个数字, 输入的前两个数字与后两个数字必须相同。

```
>>> twins = re.compile(r'(\d\d)\1')
>>> twins.findall('12341212345453999928202')
['12', '45', '99']
>>>
```

符合要求的数字只有 1212、4545、9999, 由于分组设置是 `(\d\d)` 两个数字, 而 `findall()` 以列表返回各个分组, 因此结果是 12、45、99。如果想要获取 1212、4545、9999 这样的结果, 要使用 `finditer()` 方法, 通过迭代 `Match` 并调用 `group()` 来获取符合整个正则表达式的子字符串。例如:

```
>>> for m in twins.finditer('12341212345453999928202'):
...     print(m.group())
...
1212
4545
9999
>>>
```

前文曾说明, `(["'])(^["'])*\1` 可对比出前后引号必须一致的情况, 若想找出单引号或双引号中的文字, 如下使用 `findall()` 是行不通的。

```
>>> regex = re.compile(r'(["'])(^["'])*\1')
>>> regex.findall(r'"your right brain has nothing \'left\' and your left has nothing \'right\'"')
['"', '']
>>>
```

因为 `findall()` 以列表返回各个分组, 而分组设置为 `(["'])`, 符合的是单引号或双引号, 因此列表中才会只看到 '与', 如果要找出单引号或双引号中的文字, 必须如下编写:

```
>>> import re
>>> regex = re.compile(r'(["'])(^["'])*\1')
>>> for m in regex.finditer(r'"your right brain has nothing \'left\' and your left has nothing \'right\'"'):
...     print(m.group())
...
'left'
'right'
```



```
>>>
```

如果设置了分组, `search()` 或 `match()` 在寻找到文字时也可以使用 `group()` 指定数字, 表示要获取哪个分组, 或者是使用 `groups()` 返回一个元组 (tuple), 其中包含符合要求的分组。例如:

```
>>> m = regex.search(r"your right brain has nothing 'left'")
>>> m.group(1)
"'"
>>> m.group(2)
'left'
>>> m.groups()
('"' , 'left')
>>> m.group(0)
"'left'"
>>>
```

`group(0)` 实际上等于调用 `group()` 时不指定数字, 表示整个符合正则表达式的字符串。

🎯 字符串的替换

如果要替换符合的子字符串, 可以使用正则表达式对象的 `sub()` 方法。例如将所有单引号都换成双引号。

```
>>> regex = re.compile(r"''")
>>> regex.sub('""', "your right brain has nothing 'left' and your left brain has nothing 'right'")
'your right brain has nothing "left" and your left brain has nothing "right"'
>>>
```

如果正则表达式中有分组设置, 在使用 `sub()` 时可以使用 `\num` 来捕捉被分组匹配的文字, `num` 表示第几个分组。下面示范如何将用户邮件地址从 `.com` 替换为 `.cc`。

```
>>> regex = re.compile(r'([a-zA-Z]+\d*)@([a-z]+?.)com')
>>> regex.findall('caterpillar@openhome.com')
[('caterpillar', 'openhome.')]
>>> regex.sub(r'\1@2cc', 'caterpillar@openhome.com')
'caterpillar@openhome.cc'
>>>
```

整个正则表达式匹配了 `'caterpillar@openhome.com'`, 第一个分组捕捉到 `'caterpillar'`, 第二个分组捕捉到 `'openhome.'`, `\1` 与 `\2` 就分别代表这两个部分。

下面这个范例可以让我们输入正则表达式与想对比的字符串, 执行结果将显示出匹配的结果。

regex regex.py

```
import re, sre_constants

def whereis(regex, text):
    try:
        pattern = re.compile(regex)
    except sre_constants.error as err:
        print('正则表达式有误')
        print(err.msg)
    else:
        for m in pattern.finditer(text):
            print('从索引 {} 开始到索引 {} 之间找到符合文字 {}'.format(m.start(), m.end(), m.group()))

regex = input('输入正则表达式: ')
```

```
text = input('输入要对比的文字: ')
whereis(regex, text)
```

范例的执行结果如下所示:

```
输入正则表达式: .*?foo
输入要对比的文字: xfooxxxxxxfoo
从索引 0 开始到索引 4 之间找到符合的文字 xfoo
从索引 4 开始到索引 13 之间找到符合的文字 xxxxxxfoo
```

11.4 文件与目录

在应用程序或系统管理的日常任务中,列出文件或目录、进行路径的切换、搜索文件或遍历目录等是经常性的需求,Python 的标准链接库中自然对这类基本需求提供了内建的解决方案。

11.4.1 使用 os 模块

在第 8 章介绍文件的读取、写入、修改时,曾经看到过 os 模块,除了针对文件内容进行存取的相关 API 之外,os 模块还可以针对一些与操作系统任务相关的操作,该模块提供了一些 API 可供调用,当然也包括了文件与目录管理相关的操作。

获取目录信息

在指定文件或目录时,如果不使用绝对路径,指定的文件或目录就是相对于工作目录的相对路径,通常这会是程序执行时的路径,若想知道当前的工作目录,或者切换工作目录,可以使用 os.getcwd()或 os.chdir()。

```
C:\Users\Justin>python
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> os.getcwd()
'C:\Users\Justin\'
>>> os.chdir(r'c:\workspace')
>>> os.getcwd()
'c:\workspace'
>>>
```

如果想知道指定的目录下有哪些文件或目录,那么可以使用 os.listdir(),这会使用列表返回文件与目录清单。如果想要以惰性的方式处理,那么可以使用 os.scandir(),这会返回 iterable 的 os.DirEntry 实例,该实例具有 is_dir()、is_file()等方法可供调用。例如:

```
>>> os.listdir(r'c:\workspace')
['ducktyping.js', 'helloworld.js', 'logging_demo']
>>> for entry in os.scandir(r'c:\workspace'):
...     if entry.is_file():
...         print('File:', entry.name)
...     elif entry.is_dir():
...         print('Dir:', entry.name, '')
...
File: ducktyping.js
File: helloworld.js
```

```
Dir:[ logging_demo ]
>>>
```

❶ 遍历目录

如果指定的目录下还有子目录，想要更深入地遍历，可以使用 `os.scandir()`，在迭代每个 `os.DirEntry` 实例时，使用 `is_dir()` 来看看是否为目录，如果是就递归进行下一层目录的遍历。例如：

files_dirs list_all.py

```
import os

def list_all(dir, action):
    action(dir)
    for entry in os.scandir(dir):
        fullpath = dir + '\\' + entry.name
        if entry.is_dir():
            list_all(fullpath, action)
        elif entry.is_file():
            print(fullpath)

list_all(r"c:\workspace", print)
```

对于这样的需求，Python 直接提供了 `os.walk()` 可供调用。`os.walk()` 会返回一个生成器，迭代这个生成器每次都会获取一个元组 (tuple)，先来看看元组里有什么：

```
>>> for t in os.walk(r'c:\workspace'):
...     print(t)
...
('c:\\workspace', ['logging_demo'], ['ducktyping.js', 'helloworld.js'])
('c:\\workspace\\logging_demo', ['.idea', '__pycache__'], ['basic_logger.py',
'basic_logger2.py', 'basic_logger3.py', 'config_demo.py', 'config_demo2.py',
'filter_demo.py', 'formatter_demo.py', 'handler_demo.py', 'logconf.json', 'logconf.py'])
('c:\\workspace\\logging_demo\\.idea', [], ['.name', 'encodings.xml', 'logging_demo.iml',
'misc.xml', 'modules.xml', 'workspace.xml'])
('c:\\workspace\\logging_demo\\__pycache__', [], ['logconf.cpython-35.pyc'])
>>>
```

每一次迭代的元组里有三个元素 (dirpath, dirnames, filenames)，dirpath 是字符串，代表当前遍历到哪一层目录，dirnames 是列表 (list)，代表当前目录下有哪些子目录，第三个元素也是列表，代表当前目录下有哪些文件。

`os.walk()` 会自行遍历子目录，因此若想列出指定目录下包含子目录的全部文件与目录清单，最简单的情况下，可以如下编写：

files_dirs list_all2.py

```
import os

def list_all(dir, action):
    for dirpath, dirnames, filenames in os.walk(dir):
        action(dirpath)
        for filename in filenames:
            action(dirpath + '\\' + filename)

list_all(r"c:\workspace", print)
```

执行的结果如下：


```

c:\workspace
c:\workspace\ducktyping.js
c:\workspace\helloworld.js
c:\workspace\logging_demo
c:\workspace\logging_demo\basic_logger.py
c:\workspace\logging_demo\basic_logger2.py
...
c:\workspace\logging_demo\.idea
c:\workspace\logging_demo\.idea\.name
...
c:\workspace\logging_demo\__pycache__
c:\workspace\logging_demo\__pycache__\logconf.cpython-35.pyc

```

创建、修改、删除目录

如果想要创建目录，那么可以使用 `os.mkdir()`；如果想要对目录进行更名，那么可以使用 `os.rename()`，这个函数也可以用来对文件进行更名；如果想要删除目录，那么可以使用 `os.rmdir()`；如果要删除文件，那么必须使用 `os.remove()`。

```

>>> os.listdir()
['ducktyping.js', 'files_dirs', 'helloworld.js', 'logging_demo']
>>> os.mkdir('test')
>>> os.listdir()
['ducktyping.js', 'files_dirs', 'helloworld.js', 'logging_demo', 'test']
>>> os.rename('test', 'demo')
>>> os.listdir()
['demo', 'ducktyping.js', 'files_dirs', 'helloworld.js', 'logging_demo']
>>> os.remove('demo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
PermissionError: [WinError 5] 存取被拒.: 'demo'
>>> os.rmdir('demo')
>>> os.listdir()
['ducktyping.js', 'files_dirs', 'helloworld.js', 'logging_demo']
>>> os.remove('ducktyping.js')
>>> os.listdir()
['files_dirs', 'helloworld.js', 'logging_demo']
>>>

```

这里只是关于 `os` 模块中文件与目录的简介，实际上还有其他函数可以使用。如果是在 Unix 系统上，`os` 模块还有 `chown()` 之类的 Unix 文件系统相关的函数或参数可以引用，详情可参阅“Files and Directories¹”的说明。

11.4.2 使用 `os.path` 模块

在刚才的范例中，我们使用了 `dir + '\\' + entry.name` 与 `dirpath + '\\' + filename` 来创建路径信息，这样的方式其实容易出错，而且会造成应用程序与操作系统的依赖性（以这个例子来说，就是路径分隔使用 `\\` 固定了）。

有关于路径的操作，像路径的组合、相对路径转为绝对路径、获取文件所在的目录路径等，

¹ Files and Directories: docs.python.org/3/library/os.html#files-and-directories

或者是路径对应的文件目录或目录信息，Python 提供了 `os.path` 模块来解决相关的问题。

🎯 路径计算

先从刚才提到的 `dir + '\\' + entry.name` 与 `dirpath + '\\' + filename` 问题开始，对于这样的需求，其实可以使用 `os.path.join()` 函数来解决，例如，`list_all.py` 可以改写如下：

files_dirs list_all3.py

```
import os, os.path

def list_all(dir, action):
    action(dir)
    for entry in os.scandir(dir):
        fullpath = os.path.join(dir, entry.name)
        if entry.is_dir():
            list_all(fullpath, action)
        elif entry.is_file():
            print(fullpath)

list_all(r"c:\workspace", print)
```

`os.path.join()` 函数会根据应用程序执行在哪个系统上自动判断要使用 `\` 或 `/` 来作为路径分隔符。同样的道理，刚才的 `list_all2.py` 中，在 `import os.path` 之后，`dirpath + '\\' + filename` 也可以改为 `os.path.join(dirpath, filename)`。

如果有个相对路径，想要知道它的绝对路径，可以使用 `os.path.abspath()`，如果有个文件路径，只想获取目录部分的路径，可以使用 `os.path.dirname()`，如果有个路径想要去除掉父路径，可以使用 `os.path.basename()`。

```
>>> import os.path
>>> os.path.abspath('helloworld.js')
'C:\workspace\helloworld.js'
>>> os.path.dirname(r'c:\workspace\helloworld.js')
'c:\workspace'
>>> os.path.basename(r'c:\workspace\helloworld.js')
'helloworld.js'
>>>
```

如果路径有冗余的信息，例如 `A/B`、`A/B/`、`A./B` 与 `A/foo/./B` 等，可以使用 `os.path.normpath()` 方法将其规范化（Normalize）而成为 `A/B`。例如：

```
>>> os.path.normpath(r'c:\workspace\.\helloworld.js')
'c:\workspace\helloworld.js'
>>> os.path.normpath(r'c:\workspace\..\helloworld.js')
'c:\helloworld.js'
>>> os.path.normpath(r'c:\workspace\.\.\helloworld.js')
'c:\workspace\helloworld.js'
>>>
```

如果想要计算某个路径，相对于当前工作路径下的相对关系，可以使用 `os.path.relpath()`，如果指定了第二个参数，就是计算某个路径相对于第二个参数的路径关系。例如：

```
>>> os.getcwd()
'C:\workspace'
>>> os.path.relpath(r'c:\Program Files')
'..\Program Files'
>>> os.path.relpath(r'c:\Program Files', r'c:\workspace\logging_demo')
```

```
'..\..\..\Program Files'
>>>
```

提示 >>>

`os.relpath()`不要和 `os.realpath()`搞错了，后者是在 Unix 环境下用来获取 Symbol link 的实际路径。

● 路径判定

路径只是一个位置的表示，实际上路径指向的位置不一定存在实际资源，我们可以使用 `os.path.exists()`来判定路径指向的位置是否真的存在资源。

```
>>> os.path.exists(r'c:\workspace\xyz.js')
False
>>> os.path.exists(r'c:\workspace\helloworld.js')
True
>>> os.path.isfile(r'c:\workspace\helloworld.js')
True
>>> os.path.isdir(r'c:\workspace\helloworld.js')
False
>>> os.path.isabs(r'c:\workspace\helloworld.js')
True
>>>
```

可以看到，除了测试资源是否存在，也可以使用 `os.path.isfile()`、`os.path.isdir()`来测试路径指向的资源是否为文件或目录，`os.path.isabs()`用来测试路径是否为绝对路径。

● 路径信息

如果想要获取路径指向的资源的创建时间、最后访问时间、修改时间，可以分别使用 `os.path.getctime()`、`os.path.getatime()`、`os.path.getmtime()`，返回的数字是自 epoch（纪元）起经过的秒数，可搭配 11.1 小节介绍的时间日期 API，转换为人类的可读的时间格式。例如简单转换为 UTC 时间。

```
>>> t = os.path.getatime(r'c:\workspace\helloworld.js')
>>> t
1464658255.027424
>>> import time
>>> time.asctime(time.gmtime(t))
'Tue May 31 01:30:55 2016'
>>> os.path.getsize(r'c:\workspace\helloworld.js')
375
>>>
```

在上面的范例中也看到了 `os.path.getsize()`的使用，它会返回路径指向资源的大小，单位是字节。

提示 >>>

如果需要进行文件或目录的复制，那么可以使用 `shutil` 模块，它提供了 `copyfile()`、`copy()`、`copy2()`、`copytree()`、`rmtree()`等高级操作。如果需要使用面向对象风格的文件系统路径，那么可以考虑使用 Python 3.4 之后新增的 `pathlib` 模块。

11.4.3 使用 glob 模块

如果想在工作目录下搜索文件，例如想搜索 `.py` 文件，可以使用 `glob` 模块。例如：


```
>>> import glob
>>> glob.glob('*.py')
['basic_logger.py', 'basic_logger2.py', 'basic_logger3.py', 'config_demo.py',
'config_demo2.py', 'filter_demo.py', 'formatter_demo.py', 'handler_demo.py', 'logconf.py']
>>>
```

glob 是个很简单的模块，支持简易的 Glob 模式对比语法，它比正则表达式简单，常用于目录与文件名的对比。glob()函数可使用的符号说明如表 11.7 所示。

表 11.7 glob()可使用的符号

符号	说明
*	对比全部的东西
?	对比任一字符
[seq]	对比 seq 中任一字符
[!seq]	对比 seq 以外的任一字符

glob()有个 recursive 参数，如果设置为 True，**模式可用来对比任意文件，以及 0 或多个目录或子目录，也就是说，可以有跨目录且递归地搜索子目录的效果。例如，工作目录下有 files_dirs 与 logging_demo 两个目录，如果其中各有一些.py 文件，那么可以使用**/*.py 模式同时搜索出来。

```
>>> glob.glob('**/*.py', recursive = True)
['files_dirs\\list_all.py', 'files_dirs\\list_all2.py', 'files_dirs\\list_all3.py',
'logging_demo\\basic_logger.py', 'logging_demo\\basic_logger2.py',
'logging_demo\\basic_logger3.py', 'logging_demo\\config_demo.py',
'logging_demo\\config_demo2.py', 'logging_demo\\filter_demo.py',
'logging_demo\\formatter_demo.py', 'logging_demo\\handler_demo.py',
'logging_demo\\logconf.py']
>>>
```

以下是几个 glob()可使用的对比范例。

- *.py 对比.py 结尾的字符串。
- **/*.test.py 跨目录对比 test.py 结尾的路径，在 glob()的 recursive 设置为 True 时，bookmark_test.py、command_test.py 都符合。
- ???符合三个字符，例如 123、abc 会符合。
- a?*.py 对比 a 之后至少一个字符，并以.py 结尾的字符串。
- *[0-9]*对比的字符串中要有一个数字。

glob()会以列表返回符合的路径，iglob()的功能与 glob()相同，不过返回迭代器。下面编写一个范例，指定 glob()可接受的模式来搜索指定路径下符合的文件。

```
files_dirs glob_search.py

import sys, os, glob

try:
    path = sys.argv[1]
    pattern = sys.argv[2]
except IndexError:
    print('请指定搜索路径与 glob 模式')
    print('例如: python glob_search.py c:\workspace **/*.py')
else:
    os.chdir(path)
```

```
for p in glob.iglob(pattern, recursive = True):
    print(p)
```

范例执行的结果如下：

```
C:\workspace\files_dirs>python glob_search.py c:\workspace **\*.pyc
logging_demo\__pycache__\logconf.cpython-35.pyc
```

11.5 重点复习

GMT 时间常常不严谨（且有争议性）地当成是 UTC 时间。现在 GMT 已不作为标准时间使用。在 1972 年引入 UTC 之前，GMT 与 UT 是相同的。1967 年定义的国际原子时间（IAT）将秒的国际单位定义为铯原子辐射振动 9192631770 周耗费的时间，时间从 UT 的 1958 年开始同步。

由于基于铯原子振动定义的秒长是固定的，然而地球自转会越来越慢，这会使得实际上 IAT 时间不断地超前基于地球自转的 UT 系列时间，为了保持 IAT 与 UT 时间不要差距过大，因而提出了具有折衷修正版本的世界标准时间（即世界协调时间），常简称为 UTC。UTC 经过了几次的时间修正，为了简化日后对时间的修正，1972 年 UTC 采用了闰秒修正。

Unix 系统的时间表示法定义为 UTC 时间 1970 年（Unix 元年）1 月 1 日 00:00:00 为起点而经过的秒数，不考虑闰秒修正，用以表达时间轴上某一瞬间。

Epoch（纪元）为某个特定时代的开始，时间轴上某一瞬间。

为了让人们对时间的认知符合日常作息，因而设置了 UTC 偏移，大致上来说，经度每 15 度就偏移一小时。不过有些国家的领土横跨的经度很大，一个国家采用多个时区的时间反而会造成困扰，因而不采取每 15 度偏移一小时的做法，像美国仅有 4 个时区，中国、印度采单一时区。

有些高纬度国家夏季、冬季日照时间差异很大，为了节省能源会尽量利用夏季日照，因而实施夏令时也称为夏时制。

如果想获取系统的时间，Python 的 time 模块提供了一个接口，用来调用各个平台上的 C 链接库函数，它提供的相关函数通常与 epoch 有关。

UTC 是一种绝对时间，与时区无关，也没有夏令时的问题。

time 模块提供的是低级的机器时间，也就是从 epoch 起经过的秒数，然而有些辅助函数可以做些简单的转换，以便成为人类可理解的时间概念。

人类在时间概念的表达大多是笼统、片段的信息。datetime、date、time 默认是没有时区信息的，单纯用来表示一个日期或时间概念。

datetime 实例本身默认并没有时区信息，它单纯表示本地时间。然而它也可以补上时区信息。datetime 类中的 tzinfo 类可以用继承方式来实现时区的相关信息与操作。从 Python 3.2 开始，datetime 类新增了 timezone 类，它是 tzinfo 的子类，用来提供基本的 UTC 偏移时区的实现。

若需要 timezone 以外的其他时区定义，可以额外安装 Python 社区贡献的 pytz 模块。

若必须处理时区问题，一个常见的建议是使用 UTC 进行时间的存储或操作，因为 UTC 是绝对时间，不考虑夏令时等问题，在必须使用当地时区的场合，再使用 datetime 实例的 astimezone() 进行转换。

一般来说，一个模块只需要一个 Logger 实例，因此尽管可以直接构建 Logger 实例，不过建议

通过 `logging.getLogger()` 来获取 `Logger` 实例。调用 `getLogger()` 时可以指定名称，相同名称下获取的 `Logger` 会是同一个实例。

父层级相同的 `Logger`，父 `Logger` 的配置也相同。`Logger` 有层级关系，每个 `Logger` 处理完自己的日志操作后会再委托父 `Logger` 处理。如果想要调整根 `Logger` 的配置，可以使用 `logging.basicConfig()`。

自 Python 3.2 开始，建议改用 `logging.config.dictConfig()`，可使用一个字典对象来设置配置信息。

在编写正则表达式时，建议使用原始字符串。找出并理解诠释字符想要诠释的概念，对于正则表达式的阅读非常重要。

解析、验证正则表达式往往是最耗时间的阶段，在频繁使用某正则表达式的场合，若可以将解析、验证过后的正则表达式重复使用，则可以提高效率。

`glob` 是个很简单的模块，支持简易的 `Glob` 模式对比语法，它比正则表达式简单，常用于目录与文件名的对比。

课后练习

实践题

1. 编写一个程序，可如下显示本月日历，使用 Python 内建的 `calendar` 模块可以很简单地完成此功能，不过请试着在不使用 `calendar` 模块的情况下完成。

```

      June 2016
Mo Tu We Th Fr Sa Su
          1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30

```

2. 如果有一个 HTML 文件，其中有许多 `img` 标签，且每个 `img` 标签都被 `a` 标签包裹住。例如：

```

<a href="images/EssentialJavaScript-1-1.png" target="_blank"></a>

```

请编写一个程序读取指定的 HTML 文件名，将包裹 `img` 标签的 `a` 标签去除之后保存回原文件，也就是执行程序过后，文件中如上的 HTML 要变为：

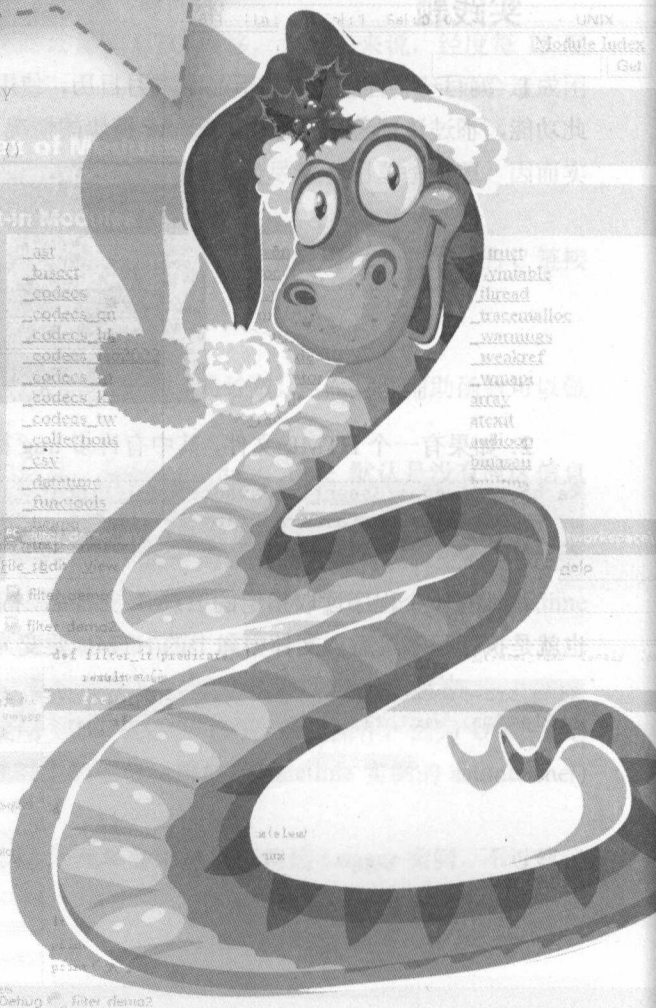
```



```


调试、测试与性能

- 使用 pdb 模块调试
- 对程序进行单元测试
- 使用 timeit 评测程序片段
- 使用 cProfile (profile) 查看评测数据



12.1 调试

在开发程序的过程中，难免因为程序编写错误而导致程序产生不正确的结果，甚至使得程序无法执行，这时我们必须找出错误并加以修正，在检测错误的时候，使用工具可以加速错误的检出，其中 **Debugger** 是最常使用也是最基本的工具之一。

12.1.1 认识 Debugger

Debugger 的使用一般来说并不困难，理由很简单，调试本身就不容易，若还学习一个不容易使用的 Debugger，岂不是更增加了调试的困难度！下面我们从第 2 章谈到的 PyCharm 中内建的 Debugger 开始介绍，并使用第 4 章实现过的 `filter_demo2.py` 作为 Debugger 的运行对象。

12.1.1.1 断点

想要在 PyCharm 中启用 Debugger，可以用鼠标右击指定的 `filter_demo2.py` 文件（也就是想要调试的源码文件），在弹出的快捷菜单中选择“Debug 'filter_demo2.py'”选项，打开 Debugger 关联的查看窗格，如图 12-1 所示，不过只是单纯地执行完程序，什么事也没发生。

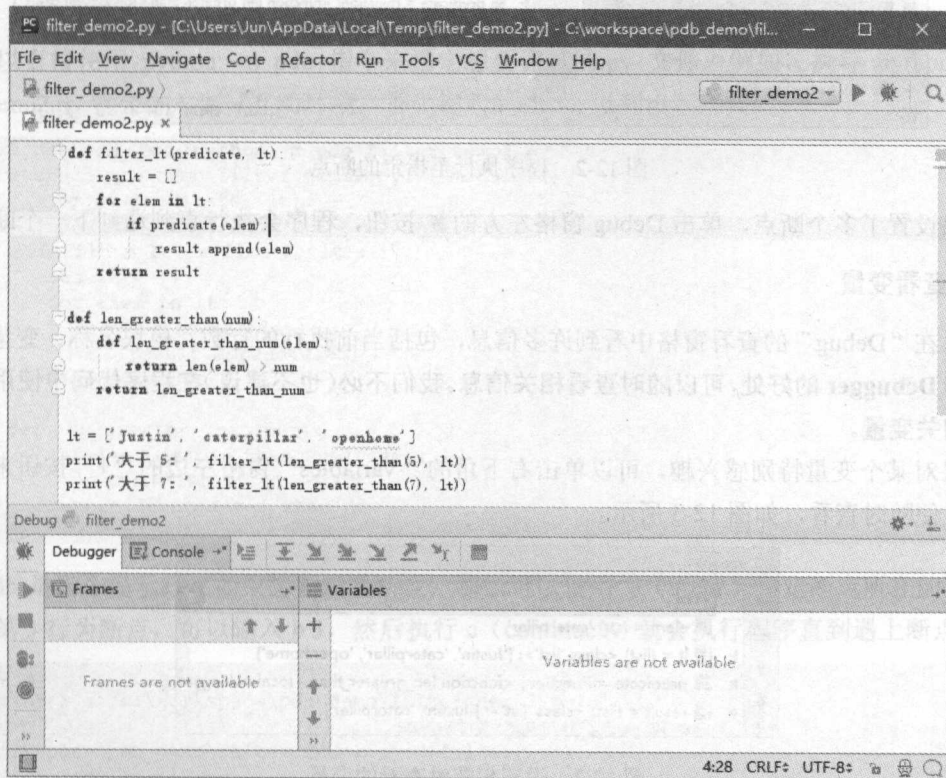


图 12-1 PyCharm 的 Debugger 关联的查看窗格

在 PyCharm 中启用 Debugger，程序会执行直到遇到断点（Break point），如果要程序执行至感兴趣的地方时停下，那么可以使用鼠标左键在程序编辑器最左边单击以设置断点，再次于源码上单

击鼠标右键选择执行“Debug”时，就会停在指定的断点外，如图 12-2 所示。

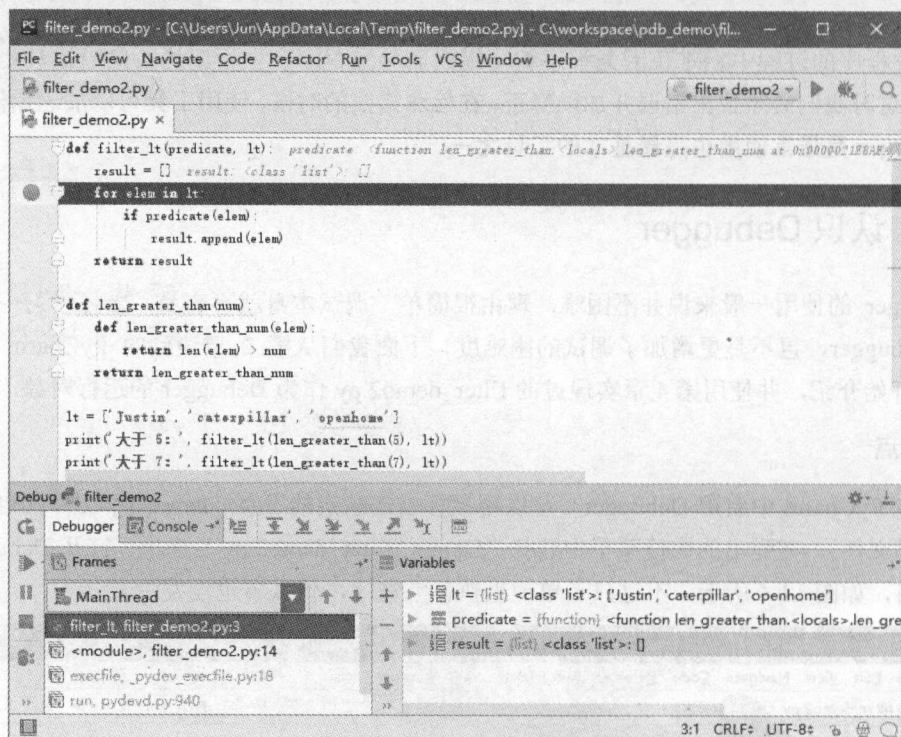



图 12-2 程序执行至指定的断点

如果设置了多个断点，单击 Debug 窗格左方的  按钮，程序会执行直到遇到下一个断点。

查看变量

可以在“Debug”的查看窗格中看到许多信息，包括当前执行的行数、模块名称、变量等，这就是使用 Debugger 的好处，可以随时查看相关信息。我们不必（也不建议）在程序代码中使用 `print()` 来查看相关变量。

如果对某个变量特别感兴趣，可以单击右下角的“Variables”窗格左边的“+”按钮来添加该变量，方便随时查看，如图 12-3 所示。

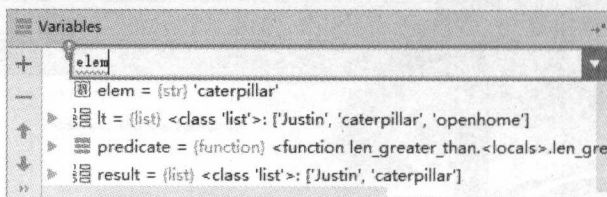





图 12-3 设置想随时查看的变量

单步执行

如果程序已经停在某个断点，想要执行下一行程序，可以使用 Debugger 的单步执行功能，不同的 Debugger 会提供不同的单步执行功能，不过基本上都会有 Step Over、Step Into、Step Out。就

Python 而言,可直接使用“Debug”窗格上的    工具栏,鼠标移至按钮上停一会儿,就会出现 Step Over、Step Into、Step Out 的字样,提示该按钮的调试功能。

Step Over 就是执行程序代码的下一步,如果下一步是个函数,会执行完该函数直至从函数返回,如果只是想看函数的执行结果或返回值是否正确,就可以使用 **Step Over**。

如果发觉函数执行结果并不正确,就可以使用 **Step Into**,如果下一步是个函数调用,就会进入函数内部单步执行,以便查看函数中的演算与每一步的执行结果。

如果当前正在某个函数之中,接下来不想单步查看函数中剩余程序代码的执行情况,就可以执行 **Step Out**,完成当前函数内未执行完的部分,并返回上一层调用函数的位置。

不同的 Debugger 会基于以上介绍的功能进行扩充。例如,PyCharm 还提供了 Step Into My Code、Run to Cursor 等功能,稍加探索一下,应该就可以理解它们的作用了。

12.1.2 使用 pdb 模块

编写 Python 程序时,如果手边正好没有集成的开发环境,只能在文本模式下执行程序进行调试,那么可以使用 Python 内建的 **pdb** 模块。

Debugger 指令

可以直接使用 `python -m pdb` 指令来指定想要调试的.py 文件。例如,执行 `python -m pdb filter_demo2.py` 就会进入调试互动环境,首先键入 `l (list)` 以列出程序代码。

```
C:\workspace\pdb_demo>python -m pdb filter_demo2.py
> c:\workspace\pdb_demo\filter_demo2.py(1)<module>()
-> def filter_lt(predicate, lt):
(Pdb) l
1  -> def filter_lt(predicate, lt):
2      result = []
3      for elem in lt:
4          if predicate(elem):
5              result.append(elem)
6      return result
7
8  def len_greater_than(num):
9      def len_greater_than_num(elem):
10         return len(elem) > num
11         return len_greater_than_num
(Pdb)
```

(Pdb)是指令提示行,如果想要设置断点,那么可以使用 `b (break)` 指定断点所在的行数,例如设置第 3 行为断点,可以输入 `b 3`,然后执行 `c (continue)`,就会执行程序直到遇上断点。

```
(Pdb) b 3
Breakpoint 1 at c:\workspace\pdb_demo\filter_demo2.py:3
(Pdb) c
> c:\workspace\pdb_demo\filter_demo2.py(3)filter_lt()
-> for elem in lt:
(Pdb)
```

被设置的断点会有个编号,例如上面可以看到 **Breakpoint 1**,表示这是第一个被设置的断点。此时如果想查看变量,那么可以使用 `p (print)` 加上变量名称,如果想执行 **Step Over**,那么可以键入 `n(next)`,如果想执行 **Step Into**,那么可以键入 `s(step)`,如果想 **Step Out**,那么可以键入 `r(return)`。

例如：

```
(Pdb) n
> c:\workspace\pdb_demo\filter_demo2.py(4)filter_lt()
-> if predicate(elem):
(Pdb) p elem
'Justin'
(Pdb) s
--Call--
> c:\workspace\pdb_demo\filter_demo2.py(9)len_greater_than_num()
-> def len_greater_than_num(elem):
(Pdb) n
> c:\workspace\pdb_demo\filter_demo2.py(10)len_greater_than_num()
-> return len(elem) > num
(Pdb) p len(elem)
6
(Pdb) r
--Return--
> c:\workspace\pdb_demo\filter_demo2.py(10)len_greater_than_num()->True
-> return len(elem) > num
(Pdb) n
> c:\workspace\pdb_demo\filter_demo2.py(5)filter_lt()
-> result.append(elem)
(Pdb)
```

执行 1 时可以看到断点与当前执行的位置，执行 1 时可以指定数字，表示从第几行开始显示源码。例如：

```
(Pdb) 1 1
1 def filter_lt(predicate, lt):
2     result = []
3 B     for elem in lt:
4         if predicate(elem):
5 ->         result.append(elem)
6     return result
7
8 def len_greater_than(num):
9     def len_greater_than_num(elem):
10         return len(elem) > num
11     return len_greater_than_num
(Pdb)
```

若要查看断点可以键入 b，使用 cl (clear) 并指定断点号码即可清除断点。

```
(Pdb) b
Num Type      Disp Enb Where
1 breakpoint keep yes at c:\workspace\pdb_demo\filter_demo2.py:3
(Pdb) cl 1
Deleted breakpoint 1 at c:\workspace\pdb_demo\filter_demo2.py:3
(Pdb) b
(Pdb)
```

1 默认只显示 11 行，可以指定显示从哪一行至哪一行，使用 unt (until) 并指定行数，可以直接执行程序到指定的行数（如果中间没有遇上断点）。例如：

```
(Pdb) 1 12, 15
12
13 lt = ['Justin', 'caterpillar', 'openhome']
14 print('大于 5: ', filter_lt(len_greater_than(5), lt))
15 print('大于 7: ', filter_lt(len_greater_than(7), lt))
(Pdb) unt 14
> c:\workspace\pdb_demo\filter_demo2.py(14)<module>()
```

```
-> print('大于 5: ', filter_lt(len_greater_than(5), lt))
(Pdb) n
大于 5: ['Justin', 'caterpillar', 'openhome']
> c:\workspace\pdb_demo\filter_demo2.py(15)<module>()
-> print('大于 7: ', filter_lt(len_greater_than(7), lt))
(Pdb)
```

如果想重新运行程序，那么可以使用 `restart`，如果想离开 `pdb`，那么可以执行 `q` (quit)。

🔍 `pdb.run()`

如果想要单独针对某个函数进行调试，那么可以使用 `pdb.run()` 函数，通常可以在 REPL 中进行这类操作。例如，若只想对 `filter_demo2.py` 中的 `filter_lt()` 进行调试，可以如下操作：

```
>>> import filter_demo2
大于 5: ['Justin', 'caterpillar', 'openhome']
大于 7: ['caterpillar', 'openhome']
>>> import pdb
>>> pdb.run('filter_demo2.filter_lt(lambda n: n < 3, [3, 1, 2, 6, 7, 4, 5])')
> <string>(1)<module>()
(Pdb) s
--Call--
> c:\workspace\pdb_demo\filter_demo2.py(1)filter_lt()
-> def filter_lt(predicate, lt):
(Pdb) p lt
[3, 1, 2, 6, 7, 4, 5]
(Pdb) n
> c:\workspace\pdb_demo\filter_demo2.py(2)filter_lt()
-> result = []
(Pdb) n
> c:\workspace\pdb_demo\filter_demo2.py(3)filter_lt()
-> for elem in lt:
(Pdb) n
> c:\workspace\pdb_demo\filter_demo2.py(4)filter_lt()
-> if predicate(elem):
(Pdb) p elem
3
(Pdb) c
>>>
```

执行 `pdb.run()` 时可以指定想要执行的程序代码，进入 (Pdb) 指令提示行之后，可以执行的指令有 `p`、`n`、`s`、`c` 等，指定的程序代码执行完毕后，才会离开 (Pdb) 指令提示行状态。

🔍 `pdb.set_trace()`

也可以将 `pdb.set_trace()` 直接编写在源码中，当程序执行到 `pdb.set_trace()` 时，就会进入 (Pdb) 指令提示行状态，这时就可以执行 Debugger 指令。直接在源码中执行 `pdb.set_trace()` 的好处是如果程序因为例外而无法继续下去，就可以使用 `pdb.pm()` 回到例外发生时的上一步，以便进行相关变量的查看。

例如，若有个 `filter_demo3.py` 如下：

```
pdb_demo filter_demo3.py
```

```
import pdb
pdb.set_trace()
```

```
def filter_lt(predicate, lt):
    result = []
```



```

    for elem in lt:
        if predicate(elem):
            result.append(elem)
    return result

def len_greater_than(num):
    def len_greater_than_num(elem):
        return len(elem) > num
    return len_greater_than_num

lt = ['Justin', 'caterpillar', 'openhome', 24]
print('大于 5: ', filter_lt(len_greater_than(5), lt))
print('大于 7: ', filter_lt(len_greater_than(7), lt))

```

在这里故意在 `lt` 中设置了一个数字，由于数字无法使用 `len()` 获取长度，因此执行时一定会发生 `TypeError` 例外。假设我们并不知道这样的问题，想要使用 `pdb.set_trace()` 来调试：

```

>>> import filter_demo3
> c:\workspace\pdb_demo\filter_demo3.py(4)<module>()
-> def filter_lt(predicate, lt):
(Pdb) c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\workspace\pdb_demo\filter_demo3.py", line 4, in <module>
    def filter_lt(predicate, lt):
  File "C:\workspace\pdb_demo\filter_demo3.py", line 7, in filter_lt
    if predicate(elem):
  File "C:\workspace\pdb_demo\filter_demo3.py", line 13, in len_greater_than_num
    return len(elem) > num
TypeError: object of type 'int' has no len()
>>> pdb.pm()
> c:\workspace\pdb_demo\filter_demo3.py(13)len_greater_than_num()
-> return len(elem) > num
(Pdb) p elem
24
(Pdb) p num
5
(Pdb) p len(elem)
*** TypeError: object of type 'int' has no len()
(Pdb)

```

可以看到，当程序因为例外中断后，执行 `pdb.pm()` 就会来到 `return len(elem) > num` 处，最后执行 `p len(elem)` 时就会发现问题所在了。

以上介绍的是 `pdb` 模块的基本使用方式，实际上 `pdb` 模块还有更多高级用法，除了在文本模式下不能使用鼠标操作外，`pdb` 的实际功能完全不输给集成开发工具上的 `Debugger`。如果有需要，那么可以再深入看看官方文档关于 `pdb` 模块¹ 的说明。

¹ `pdb` 模块：docs.python.org/3/library/pdb.html

12.2 测试

为程序编写测试程序，确保功能符合预期，一直都是确保程序质量时非常建议的方式之一，这对于身为动态类型语言（Dynamically-typing language）的 Python 来说非常重要。由于变量没有类型，如果有类型上的操作错误，基本上是在运行到该段程序代码时才会产生错误信息，不像静态类型（Statically-typing language）语言有编译程序，在程序运行前检查类型的正确性，因此在 Python 程序中检查出类型不正确的任务必须由开发者来承担，减轻这个负担的最好方式就是编写良好的测试程序。

提示 >>>

对于静态类型的程序设计语言而言，尽管有编译程序等工具来协助开发者在程序运行之前检查类型错误方面的问题，不过设计优良的测试程序来检测运行时刻功能是否符合预期也非常重要；对于动态类型的程序设计语言，现在也有一些类型注释方案提供分析工具在程序运行前检查类型信息，像 Python 3.5 中加入的 Type Hinting¹。

在 Python 的世界中，当然不乏编写测试的相关工具，例如：

- assert 语句是在程序中安插断言（Assertion）时很方便的一个方式。
- doctest 模块在程序代码中寻找类似 Python 互动环境的文字片段，执行并验证程序是否符合预期。
- unittest 模块有时称为 PyUnit，是 JUnit²的 Python 实现。
- 第三方测试工具（像 nose、pytest 等）。

12.2.1 使用 assert 断言

要在程序中安插断言，可以使用 assert，其语法如下：

```
assert_stmt ::= "assert" expression ["," expression]
```

使用 assert expression 相当于以下的程序片段：

```
if __debug__:
    if not expression: raise AssertionError
```

如果有两个 expression，例如 assert expression1、expression2，相当于以下的程序片段：

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

也就是说，第二个 expression 的结果会被当作 AssertionError 的例外信息。`__debug__` 是个内建

¹ Type Hinting: docs.python.org/3/library/typing.html

² JUnit: junit.org

变量，一般情况下是 `True`，若执行时需要优化（在执行时加上 `-O` 自变量）则会是 `False`。下面是互动环境中的一些例子。

```
C:\workspace>python
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> assert 1 == 1
>>> assert 1 != 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> __debug__
True
>>> exit()
C:\workspace>python -O
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> assert 1 != 1
>>> __debug__
False
>>>
```

那么应该什么时候使用断言呢？一般有几个建议：

- 前置条件断言在客户端调用函数之前已经准备好某些条件。
- 后置条件验证客户端调用函数之后具有函数承诺的结果。
- 类不变式（Class invariant）验证对象某个时间点下的状态。
- 内部不变式（Internal invariant）使用断言取代注释。
- 流程控制不变式（Control-flow invariant）断言程序流程中绝不会执行到的程序代码部分。

如果发现类似下面的程序片段，那么可以使用前置条件断言。

```
def __set_refresh_interval(interval):
    if interval > 0 and interval <= 1000 / MAX_REFRESH_RATE:
        raise ValueError('Illegal interval: ' + interval)
    之后是函数的程序流程
```

程序中的 `if` 检查进行了防御式程序设计（Defensive programming），检查了用户指定的参数值是否在后续执行时允许的范围内，这可用 `assert` 来替代。

```
def __set_refresh_interval(rate):
    (assert interval > 0 and interval <= 1000 / MAX_REFRESH_RATE,
     'Illegal interval: ' + interval)
    之后是函数的程序流程
```

提示 >>>

防御式程序设计有些不好的名声，不过并不是做了防御式程序设计就不好，可以参考“避免隐藏错误的防御性设计”¹ 文档。

一个内部不变式的例子如下：

¹ 避免隐藏错误的防御性设计：openhome.cc/Gossip/Programmer/DefensiveProgramming.html


```

if balance >= 10000:
    ...
elif 10000 > balance >= 100 and notVip:
    ...
else: # balance 一定少于 100
    ...

```

如果要在 `else` 的 `balance` 不少于 100 时抛出 `AssertionError` 例外，以实现速错（Fail fast）概念，而不是只使用注释来提醒开发者，那么可以改为以下编写方式：

```

if balance >= 10000:
    ...
elif 10000 > balance >= 100 and notVip:
    ...
else:
    assert balance < 100, balance
    ...

```

程序代码中有些一定不会执行到的流程区段，可以使用断言来确保这些区段被执行时抛出例外。例如：

```

if suit == Suit.club:
    ...
elif suit == Suit.diamond:
    ...
elif suit == Suit.heart:
    ...
elif suit == Suit.spade:
    ...
else: # 应该不会被执行到
    pass

```

如果 `suit` 只能被设置为枚举的 `Suit.club`、`Suit.diamond`、`Suit.heart`、`Suit.spade` 之一，那么 `else` 应该不能被执行到，如果 `else` 执行到就应该抛出例外，那么可以使用 `assert`。例如：

```

if suit == Suit.club:
    ...
elif suit == Suit.diamond:
    ...
elif suit == Suit.heart:
    ...
elif suit == Suit.spade:
    ...
else:
    assert False, suit

```

12.2.2 编写 doctest

Python 提供了 `doctest` 模块，它一方面用来测试程序代码，另一方面用来确认 `DocStrings` 的内容没有过期，它使用交互式的范例来执行验证，开发者只要为软件包编写 `REPL` 形式的文件就可以了。

举例来说，可以在 `util.py` 定义 `sorted()` 函数，并编写以下的 `DocStrings`：

doctest_demo util.py

```

import functools

def ascending(a, b): return a - b
def descending(a, b): return -ascending(a, b)

def __select(xs, compare):
    selected = functools.reduce(
        lambda slt, elem: elem if compare(elem, slt) < 0 else slt, xs)
    remain = [elem for elem in xs if elem != selected]
    return (xs if not remain
            else [elem for elem in xs if elem == selected]
                + __select(remain, compare))

def sorted(xs, compare = ascending):
    """
    sorted(xs) -> new sorted list from xs' item in ascending order.
    sorted(xs, func) -> new sorted list. func should return a negative integer,
        zero, or a positive integer as the first argument is
        less than, equal to, or greater than the second.

    >>> sorted([2, 1, 3, 6, 5])          ← ❶ DocStrings 中有 REPL 形式的执行范例
    [1, 2, 3, 5, 6]
    >>> sorted([2, 1, 3, 6, 5], ascending)
    [1, 2, 3, 5, 6]
    >>> sorted([2, 1, 3, 6, 5], descending)
    [6, 5, 3, 2, 1]
    >>> sorted([2, 1, 3, 6, 5], lambda a, b: a - b)
    [1, 2, 3, 5, 6]
    >>> sorted([2, 1, 3, 6, 5], lambda a, b: b - a)
    [6, 5, 3, 2, 1]
    """
    return [] if not xs else __select(xs, compare)

if __name__ == '__main__':    ← ❷ 直接执行.py 时条件判断式才会成立
    import doctest
    doctest.testmod()

```

若直接执行 util.py，则在范例中 `__name__` 变量会被设置为 `'__main__'` 字符串❷。这种模式经常用于为模块编写一个简单的自我测试程序，当直接执行某个.py 文件时，if 条件才会成立，测试的程序代码才会执行，而 import 该模块时，由于 `__name__` 会是模块名称，因此就不会在 import 时执行测试的程序代码。

在 DocStrings 中编写了 REPL 形式的执行范例❶，直接使用 python util.py 执行就会进行测试，若加上 -v 则会显示细节，例如：

```

C:\workspace\doctest_demo>python util.py -v
Trying:
    sorted([2, 1, 3, 6, 5])
Expecting:
    [1, 2, 3, 5, 6]
ok
Trying:
    sorted([2, 1, 3, 6, 5], ascending)
Expecting:
    [1, 2, 3, 5, 6]
ok

```

```

Trying:
  sorted([2, 1, 3, 6, 5], descending)
Expecting:
  [6, 5, 3, 2, 1]
ok
Trying:
  sorted([2, 1, 3, 6, 5], lambda a, b: a - b)
Expecting:
  [1, 2, 3, 5, 6]
ok
Trying:
  sorted([2, 1, 3, 6, 5], lambda a, b: b - a)
Expecting:
  [6, 5, 3, 2, 1]
ok
4 items had no tests:
  __main__
  __main__.__select__
  __main__.ascending
  __main__.descending
1 items passed all tests:
  5 tests in __main__.sorted
5 tests in 5 items.
5 passed and 0 failed.
Test passed.

```

把 REPL 形式的测试范例独立地编写在另一个文本文件中, 例如编写在一个 `util_doctest.txt` 中。

doctest_demo util_doctest.txt

The ``util`` module

=====

Using ``sorted``

```

>>> from util import *
>>> sorted([2, 1, 3, 6, 5])
[1, 2, 3, 5, 6]
>>> sorted([2, 1, 3, 6, 5], ascending)
[1, 2, 3, 5, 6]
>>> sorted([2, 1, 3, 6, 5], descending)
[6, 5, 3, 2, 1]
>>> sorted([2, 1, 3, 6, 5], lambda a, b: a - b)
[1, 2, 3, 5, 6]
>>> sorted([2, 1, 3, 6, 5], lambda a, b: b - a)
[6, 5, 3, 2, 1]

```

若想以程序代码方式来读取 `util_doctest.txt`, 则可以如下编写:

doctest_demo util2.py

```

if __name__ == '__main__':
    import doctest
    doctest.testfile("util_doctest.txt")

```

或者也可以直接执行 `doctest` 模块来加载测试用的文本文件以执行测试, 例如:

```

C:\workspace\doctest_demo>python -m doctest -v util_doctest.txt
Trying:
  from util import *

```



```

Expecting nothing
ok
Trying:
    sorted([2, 1, 3, 6, 5])
Expecting:
    [1, 2, 3, 5, 6]
Ok
...

```

12.2.3 使用 unittest 单元测试

unittest 模块有时也被称为“PyUnit”，是 JUnit 的 Python 语言实现，JUnit 是个 Java 实现的单元测试（Unit test）框架，单元测试指的是测试一个工作单元（a unit of work）的行为。举例来说，对于建筑桥墩而言，一个螺丝钉、一根钢筋、一条钢索甚至一公斤的水泥等都可称做一个工作单元，验证这些工作单元的行为或功能（硬度、张力等）都符合预期，方可确保最后桥墩安全无虞。

测试一个单元基本上要与其他的单元分开，否则就是在同时测试多个单元的正确性，或是多个单元之间的合作行为。就软件测试而言，单元测试通常指的是测试某个函数（或方法），给予该函数某些输入，预期该函数会产生某种输出，例如返回预期的值、产生预期的文件、新增预期的数据等。

Python 的 unittest 模块主要包括 4 个部分：

- 测试案例（Test case）测试的最小单元。
- 测试配备（Test fixture）执行一或多个测试前必要的预备资源，以及相关的清除资源操作。
- 测试软件包（Test suite）一组测试案例、测试软件包或者是两者的组合。
- 测试执行器（Test runner）负责执行测试并提供测试结果的组件。

测试案例

对于测试案例的编写，unittest 模块提供了一个基类 **TestCase**，可以继承它来创建新的测试案例。例如可为 **calc** 模块中的 **plus()**、**minus()** 函数编写测试案例。

unittest_demo calc_test.py

```

import unittest
import calc

class CalcTestCase(unittest.TestCase):
    def setUp(self):
        self.args = (3, 2)

    def tearDown(self):
        self.args = None

    def test_plus(self):
        expected = 5;
        result = calc.plus(*self.args);
        self.assertEqual(expected, result);

    def test_minus(self):
        expected = 1;

```

```

    result = calc.minus(*self.args);
    self.assertEqual(expected, result)

if __name__ == '__main__':
    unittest.main()           ← ❶ 执行测试

```

每个单元测试必须定义在一个 test 名称为开头的方法中，使用 python 执行此.py 文件时，会自动找出 test 开头的方法并执行。被测试的 calc 模块中只有两个简单的函数定义：

unittest_demo calc.py

```

def plus(a, b):
    return a + b

def minus(a, b):
    return a - b

```

由于 calc_test.py 中编写了 unittest.main()❶，若想执行单元测试，可以使用 python 直接执行 calc_test.py，执行结果如下所示：

```

C:\workspace\unittest_demo>python calc_test.py
..
-----
Ran 2 tests in 0.000s

OK

```

unittest.main()可以指定 verbosity 参数为 2，显示更详细的测试结果信息，另一个执行测试的方法是使用 unittest 模块并指定要测试的模块，例如 python -m unittest calc_test。

```

C:\workspace\unittest_demo>python -m unittest calc_test
..
-----
Ran 2 tests in 0.000s

OK

```

❷ 测试配备

如果定义了 setUp()方法，那么执行每个 test 开头的方法之前都会调用一次 setUp()，如果定义了 tearDown()方法，那么执行每个 test 开头的方法后都会调用一次 tearDown()，因此可以使用 setUp()、tearDown()来分别定义每次单元测试前后的资源创建与销毁。

❸ 测试软件包

根据测试的需求不同，我们可能想将不同的测试组合在一起，例如 CalcTestCase 中可能有数个 test_xxx 方法，如果只想将 test_plus 与 test_minus 组装为一个测试软件包，那么可以如下编写：

```

suite = unittest.TestSuite()
suite.addTest(CalcTestCase('test_plus'))
suite.addTest(CalcTestCase('test_minus'))

```

或者是使用一个列表来定义要组装的 test_xxx 方法清单。

```

tests = ['test_plus', 'test_minus']
suite = unittest.TestSuite(map(CalcTestCase, tests))

```

如果想要自动加载某个 `TestCase` 子类中所有 `test_xxx` 方法，可以如下编写：

```
suite = unittest.TestLoader().loadTestsFromTestCase(CalcTestCase)
```

可以任意组合测试，例如，将某个测试软件包与某个 `TestCase` 中的 `test_xxx` 方法组合为另一个测试软件包：

```
suite2 = unittest.TestSuite()
suite2.addTest(suite)
suite2.addTest(OtherTestCase('test_orz'))
```

也可以将许多测试软件包再全部组合为另一个测试软件包，例如若模块中定义了 `TheTestSuite()` 函数，可返回测试软件包，使用以下的方式进行组合：

```
suite1 = module1.TheTestSuite()
suite2 = module2.TheTestSuite()
alltests = unittest.TestSuite([suite1, suite2])
```

测试执行器

若想使用编写程序代码的方式，除了先前看过的 `unittest.main()` 函数之外，也可以在程序代码中直接使用 `TextTestRunner`，例如：

```
suite = (unittest.TestLoader().loadTestsFromTestCase(CalcTestCase))
unittest.TextTestRunner(verbosity=2).run(suite)
```

如果不想通过程序代码来定义，也可以在命令行中使用 `unittest` 模块来运行模块、类，甚至个别测试方法：

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

想知道 `unittest` 模块所有可用的自变量，可以使用以下指令：

```
python -m unittest -h
```

12.3 性能

性能评测（Profile）虽然是个很大的议题，然而想要知道程序的性能，必须要有适当的工具，这是本节要介绍的内容，就是看看 Python 内建模块中有哪些可用来评测性能。

12.3.1 timeit 模块

timeit 用来测量一个小程序片段的运行时间。在正式介绍 `timeit` 之前，来看一个场景，你会怎么编写程序以便“显示”以下执行结果呢？

```
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,3
3,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,6
3,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,9
3,94,95,96,97,98,99
```

以下的程序片段因为使用了 `+` 来串接字符串，在创建 `all` 时会比较缓慢吗？


```

strs = [str(num) for num in range(0, 99)]
all = ''
for s in strs:
    all = all + s + ','
all = all + '99'
print(all)

```

也许你也听过一种说法，对列表(list)使用 join()会比较快？

```

strs = [str(num) for num in range(0, 100)]
all = ','.join(strs)
print(all)

```

那么 all 的创建到底是使用+时比较快，还是 join()比较快呢？别再猜测了，直接试试使用 timeit 来评测看看：

```

>>> prog1 = '''
... all = ''
... for s in strs:
...     all = all + s + ','
... all = all + '99'
... '''
>>> prog2 = '''
... all = ','.join(strs)
... '''
>>> import timeit
>>> timeit.timeit(prog1, 'strs = [str(n) for n in range(99)]')
25.573636465720142
>>> timeit.timeit(prog2, 'strs = [str(n) for n in range(100)]')
1.4678500405096884
>>>

```

timeit()的第一个参数接受一个用字符串表示的程序片段，第二个参数是准备测试用的材料，也是用字符串表示的程序片段。timeit()在材料准备好之后，就会运行第一个参数指定的程序片段并测量时间，单位是秒，就结果看来，似乎是 join()胜出！

不过，以下却是相反的结果：

```

>>> timeit.timeit(prog1, 'strs = (str(n) for n in range(99))')
0.09589868111083888
>>> timeit.timeit(prog2, 'strs = (str(n) for n in range(99))')
0.3774917078907265
>>>

```

差别在哪里呢？在准备 strs 时两个都将[]改成了()。如果将 strs 的创建也考虑进去，那么结果就又不同了。

```

>>> prop1 = '''
... all = ''
... for s in [str(num) for num in range(0, 99)]:
...     all = all + s + ','
... all = all + '99'
... '''
>>> prop2 = '''
... all = ','.join([str(num) for num in range(0, 100)])
... '''
>>> import timeit
>>> timeit.timeit(prop1)
59.40711690576034
>>>
>>> timeit.timeit(prop2)

```

```
35.0965718301322
>>>
```

这里的重点在于，如果我们考虑的是一个程序片段，那么就不只是考虑+比较快还是 join() 比较快的问题，性能是整体程序结合之后的考虑，并不是单个元素快慢的问题，也不是凭空猜测，要有实际的评测作为依据。

timeit 默认执行程序片段 1 000 000 次，然后取平均时间，执行次数可通过 number 参数控制，下面是几个直接通过 API 运行的范例。

```
>>> timeit.timeit('strs=[str(n) for n in range(99)]', number = 10000)
0.33211180431703724
>>> timeit.timeit('strs=(str(n) for n in range(99))', number = 10000)
0.008800442406936781
>>> timeit.timeit('",".join([str(n) for n in range(99)])', number = 10000)
0.4142130435150193
>>> timeit.timeit('",".join((str(n) for n in range(99)))', number = 10000)
0.38637546380869026
>>> timeit.timeit('",".join(map(str, range(100)))', number = 10000)
0.28319832117244914
>>>
```

也可以通过命令行的指令来执行评测。

```
C:\workspace>python -m timeit '"','.join(str(n) for n in range(100))"
10000 loops, best of 3: 39 usec per loop
```

下面是个更实际的评测案例，针对 selectionSort()、insertionSort() 与 bubbleSort() 三个函数进行评测，程序中使用了 timeit.Timer()，针对各个程序片段分别创建了 Timer 实例，然而再使用 Timer 的 timeit() 指定评测次数，最后取平均值。

profile_demo sorting_prof.py

```
import timeit
repeats = 10000
for f in ('selectionSort', 'insertionSort', 'bubbleSort'):
    t = timeit.Timer('{0}([10, 9, 1, 2, 5, 3, 8, 7])'.format(f),
        'from sorting import selectionSort, insertionSort, bubbleSort')
    sec = t.timeit(repeats) / repeats
    print('{f}\t{sec:.6f} sec'.format(**locals()))
```

以下是执行的范例：

```
selectionSort 0.000033 sec
insertionSort 0.000024 sec
bubbleSort    0.000084 sec
```

12.3.2 使用 cProfile (profile)

cProfile 用来收集程序执行时的一些时间数据，提供了各种统计数据，对大多数用户来说是不错的工具，这是用 C 编写的扩展模块，在评测时有较低的额外成本，不过并不是所有系统上都提供了这个工具，profile 接口模仿了 cProfile，是用纯 Python 来实现的模块，因此有较高的互操作性。

下面是个使用 cProfile 的程序范例。

profile_demo sorting_cprof.py

```
import cProfile
import sorting
import random

l = list(range(500))
random.shuffle(l)
cProfile.run('sorting.selectionSort(l)')
```

下面是这个范例执行后的统计信息。

251503 function calls (251004 primitive calls) in 0.104 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.104	0.104	<string>:1(<module>)
124750	0.039	0.000	0.056	0.000	sorting.py:11(<lambda>)
500	0.012	0.000	0.012	0.000	sorting.py:12(<listcomp>)
499	0.007	0.000	0.007	0.000	sorting.py:14(<listcomp>)
124750	0.017	0.000	0.017	0.000	sorting.py:3(ascending)
1	0.000	0.000	0.104	0.104	sorting.py:6(selectionSort)
500/1	0.003	0.000	0.104	0.104	sorting.py:9(__select)
500	0.025	0.000	0.081	0.000	{built-in method _functools.reduce}
1	0.000	0.000	0.104	0.104	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

当中有许多字段需要解释一下。

- **ncalls**: “number of calls” 的缩写，也就是对特定函数的调用次数。
- **tottime**: “total time” 的缩写，花费在函数上的运行时间（不包括子函数调用的时间）。
- **percall**: tottime 除以 ncalls 的结果。
- **cumtime**: “cumulative time” 的缩写，花费在函数与所有子函数的时间（从调用函数至离开函数）。
- **percall**: cumtime 除以 ncalls 的结果。
- **filename:lineno(function)**: 提供程序代码执行时的位置信息。

除了直接查看 cProfile 的结果之外，也可以使用 **pstats** 对 cProfile 的结果进行各种运算与排序，这要先将 cProfile 收集的结果存储为一个文件，然后使用 pstats 加载文件，例如以下的范例分别针对 name、cumtime 与 tottime 进行排序并显示结果。

profile_demo sorting_cprof.py

```
import cProfile, pstats, random, sorting

l = list(range(500))
random.shuffle(l)
cProfile.run('sorting.selectionSort(l)', 'select_stats')

p = pstats.Stats('select_stats')
p.strip_dirs().sort_stats('name').print_stats()
p.sort_stats('cumulative').print_stats(10)
p.sort_stats('time').print_stats(10)
```


12.4 重点复习

在检测错误的时候有个顺手的工具可以加速错误的检出，其中 Debugger 是最常使用也是最基本的工具之一。

使用 Debugger 的好处是可以随时查看相关信息，不必（也不建议）在程序代码中使用 `print()` 来查看相关的变量。

Step Over 就是执行程序代码的下一步，如果下一步是个函数，会执行完该函数直至返回，如果只是想看函数的执行结果或返回值是否正确，就可以使用 Step Over。

如果发觉函数执行结果并不正确，那么可以使用 Step Into，如果下一步是个函数调用，就可以进入函数中单步执行，以便查看函数中的演算与每一步的执行结果。

如果当前正在某个函数之中，接下来不想单步查看函数中剩余程序代码的执行情况，可以执行 Step Out 完成当前函数未执行完的部分，并返回到上一层调用函数的位置。

编写 Python 程序时，如果手边正好没有集成开发环境，只能在文本模式下执行程序进行调试时，那么可以使用 Python 内建的 `pdb` 模块。

Python 中由于变量没有类型，如果有类型上的操作错误，基本上会在运行至该段程序代码时才会产生错误信息，不像静态类型的程序设计语言那样有编译程序，可以在程序运行之前检查类型的正确性，因此在 Python 程序中检查出类型不正确的任务必须由开发者来承担，减轻这个负担的最好方式就是编写良好的测试程序。

要在程序中安插断言，可以使用 `assert`。

`__debug__` 是个内建变量，一般情况下是 `True`，若执行时需要优化（在执行时加上 `-O` 自变量）则是 `False`。

一般而言 `assert` 的使用有几个建议：

- 前置条件断言在客户端调用函数之前已经准备好某些条件。
- 后置条件验证在客户端调用函数之后具有函数承诺的结果。
- 类不变式（Class invariant）验证对象某个时间点下的状态。
- 内部不变式（Internal invariant）使用断言取代注释。
- 流程控制不变式（Control-flow invariant）断言程序流程中绝不会执行到的程序代码部分。

Python 提供了 `doctest` 模块，它一方面用来测试程序代码，另一方面用来确认 DocStrings 的内容没有过期，它使用交互式的范例来执行验证，开发者只要为软件包编写 REPL 形式的文件即可。

`unittest` 模块有时亦称为“PyUnit”，是 JUnit 的 Python 语言实现，JUnit 是个 Java 实现的单元测试（Unit test）框架，单元测试指的是测试一个工作单元的行为。

测试一个单元基本上要与其他单元分开，否则就是在同时测试多个单元的正确性，或是多个单元之间的合作行为。就软件测试而言，单元测试通常指的是测试某个函数（或方法）。

`timeit` 用来测量一个小程序片段的运行时间。

性能是整体程序结合之后的考虑，并不是单个元素快慢的问题，也不是凭空猜测，要有实际的评测作为依据。

cProfile 用来收集程序执行时的一些时间数据，提供了各种统计数据，对大多数用户来说是不够的，这是用 C 编写的扩展模块，在评测时有较低的额外成本，不过并不是所有系统都提供了，profile 接口模仿了 cProfile，是用纯 Python 来实现的模块，因此有较高的互操作性。

可以使用 pstats 对 cProfile 的结果进行各种运算与排序。

课后练习

实践题

1. 在本书提供的范例的 samples/CH12/unittest_demo 文件夹中有一个 dvdlib1.py，你有办法使用 unittest 为它编写测试吗？
2. 在本书提供的范例的 samples/CH12/unittest_demo 文件夹中有一个 dvdlib2.py，你有办法使用 unittest 为它编写测试吗？与 dvdlib1.py 相比，哪个比较容易进行测试？有办法将 dvdlib1.py 重构 (Refactor)，让它变得更容易测试吗？

并发与并行

- 认识并发与并行处理
- 使用 subprocess 模块
- 使用 multiprocessing 模块



13.1 并发

到目前为止介绍过的各种范例都是单线程程序，也就是执行.py 从开始到结束只有一个流程。有时候设计程序时会想要针对不同的需求拥有多个流程，也就是所谓的多线程（Multi-thread）程序，这样的程序在执行时看起来像是并发（Concurrency），常被称为并发程序设计。

13.1.1 线程简介

如果要设计一个龟兔赛跑游戏，赛程长度为 10 步，每经过一秒，乌龟会前进一步，兔子可能前进两步或睡觉，那么该怎么设计呢？使用目前所学过的单线程程序可能会如下设计：

threading_demo tortoise_hare_race.py

```
import random

flags = [True, False]
total_step = 10
tortoise_step = 0
hare_step = 0

print('龟兔赛跑开始...')
while tortoise_step < total_step and hare_step < total_step:
    tortoise_step += 1          ← ❶ 乌龟走一步
    print('乌龟跑了 {} 步...'.format(tortoise_step))
    sleeping = flags[int(random.random() * 10) % 2]
    if sleeping:                ↑ ❷ 随机睡觉
        print('兔子睡着了 zzzz')
    else:
        hare_step += 2         ← ❸ 兔子走两步
        print('兔子跑了 {} 步...'.format(hare_step))
```

目前程序只有一个流程，就是从.py 开始到结束的流程。tortoise_step 递增 1 表示乌龟走一步❶，兔子可能随机睡觉❷，如果不是睡觉就将 hare_step 递增 2，表示兔子走了两步❸，只要乌龟或兔子其中一个走完 10 步就离开循环，表示比赛结束。

由于程序只有一个流程，因此只能将乌龟与兔子的行为混杂在这个流程中编写。为什么每次都先递增乌龟再递增兔子步数呢？这样对兔子很不公平啊！如果可以编写程序再启动两个线程，一个是乌龟线程，一个兔子线程，程序逻辑就会比较清楚。

在 Python 中，如果想在主线程以外独立设计线程，可以使用 threading 模块，例如可以在两个独立的函数中分别设计乌龟与兔子的线程。

threading_demo tortoise_hare_race2.py

```
import random, threading, time

def tortoise(total_step):      ← ❶ 乌龟的流程
    step = 0
    while step < total_step:
        step += 1
```

```

print('乌龟跑了 {} 步...'.format(step))

def hare(total_step):  ← ②兔子的流程
    step = 0
    flags = [True, False]
    while step < total_step:
        sleeping = flags[int(random.random() * 10) % 2]
        if sleeping:
            print('兔子睡着了 zzzz')
        else:
            step += 2
            print('兔子跑了 {} 步...'.format(step))

t = threading.Thread(target = tortoise, args = (10,))  ← ③创建 Thread 实例
h = threading.Thread(target = hare, args = (10,))

t.start()  ← ④启动 Thread
h.start()

```

在 `tortoise` 函数中，乌龟只要专心负责每秒走一步就可以了，不会混杂兔子的线程①。同样地，在 `hare` 函数中，兔子只要专心负责每秒睡觉或走两步就可以了，不会混杂乌龟的线程②。

当执行这个.py 时，使用了 `threading.Thread` 指定 `target` 参数为 `tortoise`③，这表示稍后执行 Thread 实例的 `start()`方法时④会调用 `tortoise()`函数，`args` 参数表示指定给函数的自变量，使用的是元组 (tuple)，由于 `tortoise()` 只有一个自变量，因此必须使用 (10,) 来表示这是单元素的元组。使用 `threading.Thread` 指定执行 `hare()`函数时也是类似的做法。

当 Thread 实例的 `start()`方法执行时，指定的函数就会独立地运行各个线程，因此执行时的结果会是：

```

乌龟跑了 1 步...
乌龟跑了 2 步...
兔子跑了 2 步...
乌龟跑了 3 步...
兔子跑了 4 步...
乌龟跑了 4 步...
兔子跑了 6 步...
乌龟跑了 5 步...
兔子睡着了 zzzz
乌龟跑了 6 步...
兔子睡着了 zzzz
乌龟跑了 7 步...
兔子睡着了 zzzz
乌龟跑了 8 步...
兔子跑了 8 步...
乌龟跑了 9 步...
兔子睡着了 zzzz
乌龟跑了 10 步...
兔子睡着了 zzzz
兔子睡着了 zzzz
兔子跑了 10 步...

```

如果真的必要，可以继承 `threading.Thread`，在 `__init__()`中调用 `super().__init__()`，并在类中定义 `run()`方法来实现线程功能，不过不建议这么做，因为这会使得线程与 `threading.Thread` 产生依赖性。以下范例与前面的范例功能相同，不过是继承 `threading.Thread` 的实现。

threading_demo tortoise_hare_race2.py

```
import random, threading, time

class Tortoise(threading.Thread):
    def __init__(self, total_step):
        super().__init__()
        self.total_step = total_step

    def run(self):
        step = 0
        while step < self.total_step:
            step += 1
            print('乌龟跑了 {} 步...'.format(step))

class Hare(threading.Thread):
    def __init__(self, total_step):
        super().__init__()
        self.total_step = total_step

    def run(self):
        step = 0
        flags = [True, False]
        while step < self.total_step:
            sleeping = flags[int(random.random() * 10) % 2]
            if sleeping:
                print('兔子睡着了 zzzz')
            else:
                step += 2
                print('兔子跑了 {} 步...'.format(step))

Tortoise(10).start()
Hare(10).start()
```

13.1.2 线程的启动与停止

尽管看来像是同时进行多个线程，实际上 **python** 解释器同一时间只允许执行一个线程，因此并不是真正的并行（**Parallel**）处理，只不过“有时候”切换速度快到人们感觉像是同时处理罢了。

❶ 阻断

所谓的“有时候”是指当前线程需要等待某个阻断作业完成时，例如等待输入输出，这时解释器会试着执行另一个线程，因此线程适用的场合之一就是非计算密集的场所，因为与其等待某个阻断作业完成，不如趁着等待的时间来执行其他线程。

例如以下这个程序可以指定网址下载网页，是不使用线程的版本。

threading_demo download.py

```
from urllib.request import urlopen

def download(url, file):
    with urlopen(url) as url, open(file, 'wb') as f:
        f.write(url.read())

urls = [
```



```

'http://openhome.cc/Gossip/Encoding/',
'http://openhome.cc/Gossip/Scala/',
'http://openhome.cc/Gossip/JavaScript/',
'http://openhome.cc/Gossip/Python/'
]

filenames = [
    'Encoding.html',
    'Scala.html',
    'JavaScript.html',
    'Python.html'
]

for url, filename in zip(urls, filenames):
    download(url, filename)

```

这个程序在每一次执行 for 循环时都会打开网络链接、发出 HTTP 请求，然后再进行文件写入等，在等待网络链接被打开、解析 HTTP 协议时很耗时，也就是进入阻断的时间较长，第一个网页下载完后，再下载第二个网页，接着才是第三个、第四个。读者可以先执行以上程序，看看在你的计算机与网络环境中会耗时多久。

如果可以在第一个网页等待网络链接、解析 HTTP 协议时就进行第二个、第三个、第四个网页的下载，那么效率会提高很多。例如：

threading_demo download2.py

```

import threading
from urllib.request import urlopen

def download(url, file):
    with urlopen(url) as url, open(file, 'wb') as f:
        f.write(url.read())

urls = [
    'http://openhome.cc/Gossip/Encoding/',
    'http://openhome.cc/Gossip/Scala/',
    'http://openhome.cc/Gossip/JavaScript/',
    'http://openhome.cc/Gossip/Python/'
]

filenames = [
    'Encoding.html',
    'Scala.html',
    'JavaScript.html',
    'Python.html'
]

for url, filename in zip(urls, filenames):
    t = threading.Thread(target = download, args = (url, filename))
    t.start()

```

这个范例在执行 for 循环时会创建新的 Thread 并启动，以进行网页下载，读者可以执行看看与上一个范例的差别有多少，这个范例花费的时间明显会少很多。

对于计算密集型的任务，使用线程不见得会提高处理效率，反而容易因为解释器必须切换线程而耗费不必要的时间成本，使得性能变差。

Daemon 线程

如果主线程中启动了额外线程，默认会等待被启动的所有线程都执行完才中止程序。如果一个 Thread 创建时指定了 daemon 参数为 True，在所有的非 Daemon 线程都结束时程序就会直接终止，不会等待 Daemon 线程执行结束，如果我们需要在后台执行一些常驻任务，可以指定 daemon 参数为 True。

安插线程

如果 A 线程正在运行，线程中允许 B 线程加入，等到 B 线程执行完毕后再继续 A 线程的执行，那么可以使用 join() 方法完成这个需求。这就好比你有份工作正在进行，老板又安排了另一项工作而且要求先做好，然后再进行原来正在进行的工作。

当线程使用 join() 加入至另一线程时，另一线程会等待被加入的线程工作完成之后再继续原线程的运行。join() 的意思是将线程加入另一线程的执行当中。

threading_demo join_demo.py

```
import threading

def demo():
    print('Thread B 开始...')
    for i in range(5):
        print('Thread B 执行...')
    print('Thread B 将结束...')

print('Main thread 开始...')
tb = threading.Thread(target = demo)
tb.start()
tb.join(); # Thread B 加入 Main thread 流程

print('Main thread 将结束...')
```

程序启动后主线程就开始，在主线程中新建 tb，并在启动 tb 后将之加入 (join()) 主线程的执行中，所以 tb 会先执行完毕，主线程才会再继续原来的线程，执行结果如下：

```
Main thread 开始...
Thread B 开始...
Thread B 执行...
Thread B 执行...
Thread B 执行...
Thread B 执行...
Thread B 执行...
Thread B 将结束...
Main thread 将结束...
```

如果程序中 tb 没有使用 join() 将之加入主线程的执行中，那么最后一行显示 "Main thread 将结束..." 的语句会先执行完毕。

有时候加入的线程可能处理太久，而我们不想无止境地等待这个线程执行完毕，可以在 join() 时指定时间，例如 join(10)，这表示加入的线程最多可处理 10 秒，数字可以是浮点数，如果加入的线程在指定的时间内还没执行完毕就不理它了，当前线程可继续执行自己原来的工作。

停止线程

如果要停止线程，那么必须自行实现，让线程跑完应有的流程。例如有一个线程在循环中执行某个操作，那么停止线程的方式就是让它有机会离开循环体。

threading_demo stop_demo.py

```
import threading, time

class Some:
    def __init__(self):
        self.is_continue = True

    def terminate(self):
        self.is_continue = False

    def run(self):
        while self.is_continue:
            print('running...running')
            print('bye...bye...')

s = Some()
t = threading.Thread(target = s.run)
t.start()
time.sleep(2) # 主线程停 2 秒
s.terminate() # 停止线程
```

在这个程序片段中，如果线程执行了 `run()` 方法，就会进入 `while` 循环，想要停止线程，可以调用 `Some` 的 `terminate()`，这会将 `is_continue` 设为 `False`，在跑完此次 `while` 循环，下次 `while` 条件测试为 `False` 时就会离开循环，执行完 `run()` 方法，线程也就结束了。

因此停止线程必须自行根据条件来实现，线程的暂停、重启，也必须根据需求来实现。

13.1.3 竞争、锁定、死锁

如果线程之间不需要共享数据，或者共享的数据是不可变动（Immutable）的类型，事情会简单一些。然而线程之间经常要共享一些可变动状态的数据。

竞争

如果线程之间需要共享的是可变动状态的数据，就有可能发生竞争条件（Race condition），例如有个程序范例如下：

threading_demo race_demo.py

```
import threading

def setTo1(data):
    while True:
        data['Justin'] = 1
        if data['Justin'] != 1:
            raise ValueError('setTo1 数据不一致: {}'.format(str(data)))

def setTo2(data):
```



```

while True:
    data['Justin'] = 2
    if data['Justin'] != 2:
        raise ValueError('setTo2 数据不一致: {}'.format(str(data)))

data = {}

t1 = threading.Thread(target = setTo1, args = (data, ))
t2 = threading.Thread(target = setTo2, args = (data, ))

t1.start()
t2.start()

```

在这个范例中, t1 与 t2 线程分别执行了 setTo1() 与 setTo2() 函数, 两个函数会对同一个字典(dict) 对象进行设置, 一个将 'Justin' 对应值设为 1, 另一个设为 2, 执行时我们会发现 if 检查的部分是可能成立的, 因而会抛出例外, 抛出例外的可能是 setTo1() 函数, 也可能是 setTo2() 函数, 引发例外的时间不一定, 纯粹看运气。

问题的根源在于同一时间解释器只允许一个线程执行, 它会在线程之间进行切换, 切换的时间点我们无法预测, 如果 t1 在 setTo1() 函数中 data['Justin'] = 1 执行完后解释器切换至 t2, 这时正好执行了 setTo2() 函数的 data['Justin'] = 2, 刚好不巧地, 切换再度发生而执行 t1, setTo1() 的下一句 data['Justin'] != 1 的判断成立, 因而引发例外。

🔒 锁定

如果要避免竞争的情况发生, 就必须在资源被变更与取用时的关键程序代码中实现锁定机制, 例如:

threading_demo lock_demo.py

```

import threading

def setTo1(data, lock):
    while True:
        lock.acquire()
        try:
            data['Justin'] = 1
            if data['Justin'] != 1:
                raise ValueError('setTo1 数据不一致: {}'.format(str(data)))
        finally:
            lock.release()

def setTo2(data, lock):
    while True:
        lock.acquire()
        try:
            data['Justin'] = 2
            if data['Justin'] != 2:
                raise ValueError('setTo2 数据不一致: {}'.format(str(data)))
        finally:
            lock.release()

lock = threading.Lock()
data = {}

t1 = threading.Thread(target = setTo1, args = (data, lock))

```

```
t2 = threading.Thread(target = setTo2, args = (data, lock))

t1.start()
t2.start()
```

`threading.Lock` 实例只会有两种状态，即锁定与未锁定。在非锁定状态下，可以使用 `acquire()` 方法使之进入锁定状态，此时如果再度调用 `acquire()` 方法，就会被阻断，直到其他地方调用了 `release()` 使得 `Lock` 对象成为未锁定状态，如果 `Lock` 对象不是在锁定状态，调用 `release()` 会引发 `RuntimeError` 错误。

因此对于上面的范例来说，若 `t1` 执行了 `setTo1()` 函数的 `lock.acquire()`，之后线程切换至 `t2`，执行至 `setTo2()` 函数的 `lock.acquire()` 时，由于 `lock` 处于锁定状态，因此 `t2` 被阻断，只有在线程切换回 `t1` 并执行完 `lock.release()`，使 `lock` 成为未锁定状态，`t2` 才有机会执行 `lock.acquire()` 获取锁定，然后执行后面的程序代码。

反过来，只有在线程切换回 `t2` 并执行完 `lock.release()`，使 `lock` 成为未锁定状态，`t1` 才有机会执行 `lock.acquire()` 获取锁定，然后执行后面的程序代码。因此无论是哪个线程，都能确保 `lock` 在锁定状态期间执行完关键的程序代码区域，而不会发生竞争条件。

实际上，`threading.Lock` 实现了 7.2.3 小节谈到的上下文管理器协议（Context Management Protocol），可以搭配 `with` 来简化 `acquire()` 与 `release()` 的调用，因此上面的范例也可以改写如下：

threading_demo lock_demo2.py

```
import threading

def setTo1(data, lock):
    while True:
        with lock:
            data['Justin'] = 1
            if data['Justin'] != 1:
                raise ValueError('setTo1 数据不一致: {}'.format(str(data)))

def setTo2(data, lock):
    while True:
        with lock:
            data['Justin'] = 2
            if data['Justin'] != 2:
                raise ValueError('setTo2 数据不一致: {}'.format(str(data)))

lock = threading.Lock()
data = {}

t1 = threading.Thread(target = setTo1, args = (data, lock))
t2 = threading.Thread(target = setTo2, args = (data, lock))

t1.start()
t2.start()
```

❶ 死锁

由于线程无法获取锁定时会造成阻断，因此不正确地使用 `Lock` 有可能造成性能低下，另一问题就是死锁（Dead lock），例如有些资源在多线程下彼此交叉引用，就有可能造成死锁，下面是个简单的例子：

threading_demo deadlock_demo.py

```

import threading

class Resource:
    def __init__(self, name, resource):
        self.name = name
        self.resource = resource
        self.lock = threading.Lock()

    def action(self):
        with self.lock:
            self.resource += 1
            return self.resource

    def cooperate(self, other_res):
        with self.lock:
            other_res.action()
            print('{} 整合 {} 的资源'.format(self.name, other_res.name))

def cooperate(a, b):
    for i in range(10):
        a.cooperate(b)

res1 = Resource('resource 1', 10)
res2 = Resource('resource 2', 20)

t1 = threading.Thread(target = cooperate, args = (res1, res2))
t2 = threading.Thread(target = cooperate, args = (res2, res1))

t1.start()
t2.start()

```

上面这个程序会不会发生死锁也是概率问题，读者可以尝试执行几次，有时程序可顺利地执行完成，有时程序会整个停顿。

会发生死锁的原因在于 t1 在调用 a.cooperate(b) 时，res1 被 lock（就是锁定状态），若此时 t2 正好也调用 a.cooperate(b)，会将 res2 lock（就是锁定状态），凑巧 t1 现在打算运用传入的 res2 调用 action()，结果试图调用 res2 的 lock 的 acquire() 时被阻断，而接下来 t2 打算运用传入的 res1 调用 action()，结果试图调用 res1 的 lock 的 acquire() 时也被阻断，两个线程都被阻断了。

这个范例为何有时会死锁，就是因为偶而会发生两个线程都处于“你不解除 res1 上的锁定，我就不放开 res2 的锁定”的状态。

13.1.4 等待与通知

除了基本的 threading.Lock 之外，threading 模块中还提供了其他锁定机制，例如 threading.RLock 实现了可重入锁（Reentrant lock），同一线程可以重复调用同一个 threading.RLock 实例的 acquire() 而不会被阻断，不过要注意的是，release() 时也要有对应于 acquire() 的次数，才可以完全解除锁定，threading.RLock 也实现了上下文管理器协议，可搭配 with 来使用。

Condition

另一个经常使用的锁定机制是 `threading.Condition`，正如其名称所提示的，某个线程在通过 `acquire()` 获取锁定之后，若需要在特定条件符合之前的等待，可以调用 `wait()` 方法，这会释放锁定，如果其他线程的运行促成特定条件成立，就可以调用同一 `threading.Condition` 实例的 `notify()`，通知等待条件的一个线程可获取锁定（也许有其他线程也正在等待），如果等待线程获取锁定，就会从上次调用 `wait()` 方法的地方继续执行。

`notify()` 通知等待条件的一个线程，无法预期是哪一个线程会被通知，如果等待线程有多个，还可以调用 `notify_all()`，这会通知全部等待线程去争夺锁定。

`wait()` 可以使用浮点数指定超时，单位是秒，如果等待超过指定的时间，就会自动尝试获取锁定并继续执行，`notify()` 可以指定通知的线程数量，当前的实现是指定多少就通知多少个线程（如果线程数量足够），不过文档上注明依赖于特定的数量并不安全，未来的实现可能会根据情况来通知至少指定数量以上的线程。

`wait()`、`notify()` 或 `notify_all()` 应用的常见范例就是生产者与消费者。生产者会将生产的产品交给店员，而消费者从店员处买走产品消费，但店员一次只能存储固定数量的产品。若生产者生产速度较快，店员可存储产品的量已满，店员会叫生产者等一下（`wait`），如果有空位放产品了再通知（`notify`）生产者继续生产；如果消费者速度较快，将店中产品消费完了，店员会告诉消费者等一下（`wait`），如果店中有产品了再通知（`notify`）消费者前来消费。

下面举个最简单的范例，假设生产者每次生产一个整数交给店员，消费者从店员处买走整数。

threading_demo condition_demo.py

```
import threading

def producer(clerk):
    for product in range(10):
        clerk.purchase(product)
    print('店员进货 {}'.format(product))

def consumer(clerk):
    for product in range(10):
        print('店员卖出 {}'.format(clerk.sellout()))

class Clerk:
    def __init__(self):
        self.product = -1
        self.cond = threading.Condition()

    def purchase(self, product):
        with self.cond:
            while self.product != -1:
                self.cond.wait()
            self.product = product
            self.cond.notify()

    def sellout(self):
        with self.cond:
            while self.product == -1:
                self.cond.wait()
```

← ① 产生整数给店员
 ← ② 从店员处买走整数
 ← ③ 只持有一个产品，-1 表示没有产品
 ← ④ 创建 Condition 对象
 ← ⑤ 看看店员有没有空间收储产品，如果没有就稍候
 ← ⑥ 通知等待线程
 ← ⑦ 看看当前店员有没有货，如果没有就稍候

```

p = self.product
self.product = -1
self.cond.notify()    ← ⑧ 通知等待线程
return p

```

```

clerk = Clerk();
threading.Thread(target = producer, args = (clerk, )).start()
threading.Thread(target = consumer, args = (clerk, )).start()

```

`producer()`函数使用 `for` 循环产生整数①，`clerk` 代表店员，可通过 `purchase()`方法将生产的整数设置给店员；`consumer()`函数亦使用 `for` 循环来消费整数②，可通过 `clerk` 的 `sellout()`方法从店员身上取走整数。

`Clerk` 只能持有一个整数，-1 表示当前没有产品③，`Clerk` 中创建了 `Condition` 实例来控制等待与通知④。

假设现在 `producer()`中调用了 `purchase()`，此时不会进入 `while` 循环体而等待，所以设置 `Clerk` 的 `product` 被设为指定的整数，由于此时没有线程等待，所以调用 `notify()`没有作用⑤，假设 `producer()`中再次调用 `purchase()`，此时 `Clerk` 的 `product` 不为-1，表示店员无法收货了，进入 `while` 循环，执行了 `wait()`⑥，于是执行 `producer()`的线程释放锁定进入等待。

假设 `consumer()`中调用了 `sellout()`，由于 `Clerk` 的 `product` 不为-1，因此不会进入 `while` 循环体，于是 `Clerk` 准备交货，并将 `product` 设为-1，表示货品被买走，接着调用 `notify()`通知等待线程⑧，最后将 `p` 返回，如果 `Consumer` 又调用了 `sellout()`，此时 `product` 为-1，表示没有产品了，于是进入 `while` 循环体，执行 `wait()`后进入等待⑦，若此时执行 `producer()`的线程获取锁定，则从 `purchase()`中的 `wait()`处继续执行。

生产者会生产 10 个整数，而消费者会消耗 10 个整数，虽然生产与消费的速度不一，但是由于店员处只能放置一个整数，因此只能每生产一个才消耗一个。

```

店员进货 (0)
店员卖出 (0)
店员进货 (1)
店员卖出 (1)
店员进货 (2)
店员卖出 (2)
店员进货 (3)
店员卖出 (3)
店员进货 (4)
店员卖出 (4)
店员进货 (5)
店员卖出 (5)
店员进货 (6)
店员卖出 (6)
店员进货 (7)
店员卖出 (7)
店员进货 (8)
店员卖出 (8)
店员进货 (9)
店员卖出 (9)

```

实际上，如果需要这种一进一出在线程之间交换数据的方式，`Python` 标准链接库中提供了 `queue.Queue`，在创建实例时可指定容量，这是一个先进先出的数据结构，实现了必要的锁定机制，

可以使用 `put()` 将数据置入，使用 `get()` 取走数据，因此上面的范例也可以改写为以下程序而执行结果不变。

threading_demo queue_demo.py

```
import threading, queue

def producer(clerk):
    for product in range(10):
        clerk.put(product)
        print('店员进货 {}'.format(product))

def consumer(clerk):
    for product in range(10):
        print('店员卖出 {}'.format(clerk.get()))

clerk = queue.Queue(1);
threading.Thread(target = producer, args = (clerk,)).start()
threading.Thread(target = consumer, args = (clerk,)).start()
```

标准链接库中除了 `threading` 的 `Lock`、`RLock`、`Condition` 之外，还有 `Semaphore` 与 `Barrier`。

Semaphore 与 Barrier

`Semaphore` 这个单词的意思是“信号”，创建 `Semaphore` 可指定计数器的初始值，每调用一次 `acquire()`，计数器值递减 1，在计数器为 0 时如果调用了 `acquire()`，线程就会被阻断，每调用一次 `release()`，计数器值递增 1，如果 `release()` 前计数器为 0，而且有线程正在等待，在 `release()` 并递增计数器之后，会通知等待线程。

`Barrier` 这个单词的意思是“栅栏”，顾名思义，可以设置一个栅栏并指定数量，如果有线程先来到这个栅栏，它必须等待其他线程也来到这个栅栏，直到指定的线程数量达到，全部线程才能继续往下执行。

`threading` 的文档中有个简单易懂的程序片段，如果执行服务端线程与客户端线程都必须准备就绪才能继续往下执行，可以如下编写程序：

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```


13.2 并行

如果某个任务是计算密集型的，使用线程不见得能加快执行效率，还有可能变得更缓慢。针对对于计算密集型的运算，如果能在一个新的进程（Process）并行（Parallel）运行，在如今计算机普遍都有多个核心处理器的情况下，就有机会运行得更快一些。

13.2.1 使用 subprocess 模块

subprocess 模块可以让我们在执行 Python 程序的过程中产生新的子进程。举例来说，若想在执行 Python 程序的过程中调用 Windows 文本模式下的 echo 指令获取 %date%、%time% 之类的环境变量，可以如下执行：

```
>>> import subprocess
>>> subprocess.run('echo %date%', shell = True)
2016/06/15 周三
CompletedProcess(args='echo %date%', returncode=0)
>>> subprocess.run(['echo', '%time%'], shell = True)
15:05:59.52
CompletedProcess(args=['echo', '%time%'], returncode=0)
>>>
```

🔍 subprocess.run()

从 Python 3.5 开始，建议使用 run() 函数来调用子进程，由于打算调用文本模式下的指令，shell 参数必须设为 True，run() 的第一个自变量可以接受字符串，或者是一个列表（list），列表中的元素是指令以及相关自变量。

注意 >>>

在 Python 3.5 之前，subprocess 模块有些旧式的 API，如 call()、check_all()、check_output() 等，Python 3.5 的 run() 取代了这些 API，如果想知道这些 API 分别如何使用 run() 来取代，可以参考“Replacing Older Functions with the subprocess Module¹”。

subprocess.run() 执行之后会返回 CompletedProcess 实例，而不是我们看到的“2016/06/15 周三”或“15:05:59.52”的显示结果，这是因为 echo %date% 或 echo %time% 的结果被直接送到了当前的 REPL 的标准输出。

如果想获取标准输出的执行结果，那么可以指定 run() 的 stdout 参数为 subprocess.PIPE，这会将指令的执行结果转接至 Python 程序内部，稍后就可以通过 CompletedProcess 实例的 stdout 来进行读取，例如：

```
>>> p = subprocess.run(['echo', '%time%'], shell = True, stdout = subprocess.PIPE)
```

¹ Replacing Older Functions with the subprocess Module: docs.python.org/3.5/library/subprocess.html#replacing-older-functions-with-the-subprocess-module

```
>>> p.stdout
b'15:27:43.52\r\n'
>>>
```

如果子进程必须接收标准输入，例如有个简单的 hi.py 如下：

subprocess_demo hi.py

```
name = input('your name?')
print('Hello, ' + name)
```

那么在执行 run() 时可以指定 input 参数，作为子进程的标准输入值。例如：

```
>>> p = subprocess.run(['python', 'hi.py'], input = b'Justin\n', stdout = subprocess.PIPE)
>>> p.stdout
b'your name?Hello, Justin\r\n'
>>>
```

subprocess.Popen()

subprocess.run() 的底层是通过 subprocess.Popen() 实现出来的，Popen() 有着数量庞大的参数，因而通过它可以掌握更多子进程的细节。

举例来说，刚才的范例执行结果也可以如下使用程序来实现：

```
>>> data = b'Justin\n'
>>> p = subprocess.Popen(['python', 'hi.py'], stdin = subprocess.PIPE, stdout = subprocess.PIPE)
>>> result = p.communicate(input=data)
>>> result
(b'your name?Hello, Justin\r\n', None)
>>> result[0]
b'your name?Hello, Justin\r\n'
>>>
```

Popen() 返回一个 Popen 实例，可以通过它的 communicate() 指定 input，以提供输入给子进程，communicate() 返回一个元组 (tuple)，分别是标准输出与标准错误的结果，因为这里没有指定标准错误，因此元组的第二个元素是 None。

实际上，run() 的底层也只是直接将 input 参数的值传给 Popen 对象的 communicate() 方法，而通过 subprocess.run() 执行程序时会等待子进程完成，就是因为调用了 communicate() 方法，communicate() 方法完成才会返回 CompletedProcess 实例。

也就是说，通过 subprocess.Popen() 执行程序会立即返回 Popen 实例，若真的想等待子进程结束，除了调用 communicate() 方法之外，还可以调用 Popen 实例的 wait() 方法。

不过，通过 subprocess.Popen() 执行程序会立即返回 Popen 实例，不会等待子进程结束，这就给了我们一个通过子进程进行并行处理的机会。举个例子来说，data1.txt、data2.txt、data3.txt 等文件当中有许多字符，我们必须进行一些处理。

subprocess_demo one_process.py

```
import sys

def foo(filename):
    with open(filename) as f:
        text = f.read()
```

```

ct = 0
for ch in text:
    n = ord(ch.upper()) + 1
    if n == 67:
        ct += 1
return ct

count = 0
for filename in sys.argv[1:]:
    count += foo(filename)

print(count)

```

这个范例的 data1.txt、data2.txt、data3.txt 等文件当中，实际上是随机产生的一堆字符，每个将近 10MB 容量，foo()函数的处理也只是个示范，无趣地做些转换为大写、相加、判断、计数的工作（更实际的任务可能是解压缩、正则表达式对比这类的工作），重点在于这个程序是按序的，如果指定的文件极多，那么处理起来会很低效，在多核心处理器的计算环境中也没有善用多核心的计算优势。

我们可以编写程序使用多个子进程来完成相同的工作。

subprocess_demo multi_process.py

```

import sys, subprocess

ps = [
    subprocess.Popen(
        ['python', 'one_process.py', filename],
        stdout=subprocess.PIPE
    ) for filename in sys.argv[1:]
]

count = 0
for p in ps:
    count += int(p.stdout.read())

print(count)

```

这个程序直接使用了刚才的 one_process.py，然而每次启用一个子进程只指定一个.txt 文件给 one_process.py，每个子进程各自独立运行完的结果可以通过 Popen 实例的 stdout 来读取（记得要调用其 read()，这与 CompletedProcess 实例的方式不同），转为整数后进行相加，就是我们要的结果，然而效率上会高得多。

13.2.2 使用 multiprocessing 模块

如果想要以子进程来执行函数，类似 threading 模块的 API 接口，那么可以使用 multiprocessing 模块。举个例子来说，先前的 one_process.py 可以改写如下：

multiprocessing_demo multi_process.py

```

import sys, multiprocessing

def foo(filename, queue):
    with open(filename) as f:

```



```

text = f.read()

ct = 0
for ch in text:
    n = ord(ch.upper()) + 1
    if n == 67:
        ct += 1
queue.put(ct)  ← ❶ 将结果置入 Queue 中

if __name__ == '__main__':  ← ❷ 必要的模块名称测试
    queue = multiprocessing.Queue()  ← ❸ 存储与取得结果用的 Queue
    ps = [multiprocessing.Process(target = foo, args = (filename, queue))
           for filename in sys.argv[1:]]  ← ❹ 创建 Process 对象
    for p in ps:  ← ❺ 启动 Process
        p.start()
    for p in ps:  ← ❻ 等待全部 Process 完成
        p.join()

count = 0
while not queue.empty():  ← ❼ 从 Queue 中取得全部结果
    count += queue.get()
print(count)

```

首先必须注意的是,为了在子进程执行时让 python 解释器安全地导入 main 模块,if `__name__ == '__main__'` 的测试是必要的❷,如果没有这行语句,就会引发 `RuntimeError` 错误。

为了能在进程之间进行数据的交换,这里使用了 `multiprocessing.Queue`❸,在创建 `Process` 实例时❹,API 与 `threading.Thread` 是类似的,在指定 `target` 为 `foo` 的同时,args 的指定也包含了 `Queue`,在 `foo()` 函数中执行的结果通过 `Queue` 的 `put()` 方法置入❶。

要启动 `Process` 可以调用它的 `start()` 方法❺,因为对这个范例来说,我们必须获取全部进程执行的结果,因此必须等待全部进程完成,这时可以使用 `join()` 方法❻,这会让程序执行停下,等待进程完成再继续下一步,如果不需要等待全部进程完成,例如各个进程完成的结果直接保存在各自的文件中,那么就不用使用 `join()`。

当全部进程都完成后,获取 `Queue` 中全部的结果,可以使用 `empty()` 测试 `Queue` 是否为空,使用 `get()` 来获取 `Queue` 中的元素。

注意>>>

使用 `multiprocessing` 模块时,除了 if `__name__ == '__main__'` 的测试是必要的之外,还有其他必须遵守的规范,详情可参考“Programming guidelines¹”。

虽然建议在使用 `multiprocessing` 模块时不要共享状态,然而有时进程之间难免需要进行沟通,`multiprocessing.Queue` 在线程与进程之间实现了安全、必要的锁定机制,这表示我们可以自行进行锁定。可以通过 `multiprocessing` 模块的 `Lock`、`RLock`、`Semaphore` 等来实现。

举个例子来说,若有一个程序范例如下:

¹ Programming guidelines: docs.python.org/3.5/library/multiprocessing.html#multiprocessing-programming

multiprocessing_demo no_lock.py

```
from multiprocessing import Process

def f(i):
    print('hello world', i)
    print('hello world', i + 1)

if __name__ == '__main__':
    for num in range(100):
        Process(target=f, args=(num,)).start()
```

如果想要的是标准输出每次都连续输出 i 与 $i+1$ ，这个范例不一定能达到目的，也许会有如下的输出：

```
略...
hello world 22
hello world 23
hello world 78
hello world 63
hello world 64
略...
```

显然地，78 之前或之后并没有连续的数字，这是因为各个进程竞争标准输出的关系，若想在执行时锁定某个程序片段，可以如下编写：

multiprocessing_demo lock_demo.py

```
from multiprocessing import Process, Lock

def f(lock, i):
    lock.acquire()
    try:
        print('hello world', i)
        print('hello world', i + 1)
    finally:
        lock.release()

if __name__ == '__main__':
    lock = Lock()
    for num in range(100):
        Process(target=f, args=(lock, num)).start()
```

可以看到，使用方式与 `threading.Lock` 是类似的，`multiprocessing.Lock` 也实现了上下文管理器协议，因此也可以搭配 `with` 来使用。

multiprocessing_demo lock_demo2.py

```
from multiprocessing import Process, Lock

def f(lock, i):
    with lock:
        print('hello world', i)
        print('hello world', i + 1)

if __name__ == '__main__':
```

```
lock = Lock()

for num in range(100):
    Process(target=f, args=(lock, num)).start()
```

multiprocessing 模块也提供了一些不同于 threading 模块的 API，例如 Pool，这可以创建一个工作者池（worker pool），利用 Pool 实例可以派送任务并获取 multiprocessing.pool.AsyncResult 实例，在任务完成后获取结果。

例如，先前的 multi_process.py 可以改为以下不使用 multiprocessing.Queue 的版本。

multiprocessing_demo multi_process2.py

```
import sys, multiprocessing

def foo(filename):
    with open(filename) as f:
        text = f.read()

    ct = 0
    for ch in text:
        n = ord(ch.upper()) + 1
        if n == 67:
            ct += 1
    return ct

if __name__ == '__main__':
    filenames = sys.argv[1:]
    with multiprocessing.Pool(2) as pool:  ← ①创建有两个工作者的 Pool 实例
        results = [pool.apply_async(foo, (filename,))  ← ②派送任务
                    for filename in filenames]
        count = sum(result.get() for result in results)
        print(count)
        ↑
        ③取得结果
```

在这里故意只创建了有两个工作者的 Pool 实例①，如果指定的文件超过两个，工作者任务完成后，就会自行获取下一个任务，要派送任务可以使用 Pool 的 apply_async()②，就如方法名称所提示的，这是个异步的方法，执行后会立即返回一个 multiprocessing.pool.AsyncResult 实例，通过 get()方法可以获取结果，如果任务尚未完成③，get()会阻断且等待任务完成，就这个范例来说，这就是我们需要的结果。

提示 >>>

对于异步的任务来说，可以使用高级的 concurrent.futures 模块，当中提供了 ThreadPoolExecutor 与 ProcessPoolExecutor 等 API，前者使用线程进行异步任务，后者使用进程，实际上 ProcessPoolExecutor 底层使用了 multiprocessing 模块来实现。

13.3 重点复习

在 Python 中，如果想在主线程以外独立设计线程，可以使用 threading 模块。当 Thread 实例的 start()方法执行时，指定的函数就会像是独立地运行的各自线程。

可以继承 `threading.Thread`，在 `__init__()` 中调用 `super().__init__()`，并在类中定义 `run()` 方法来实现在线程功能，不过不建议这么做，因为这会使得线程与 `threading.Thread` 产生依赖性。

实际上，python 解释器同一时间只允许执行一个线程，因此并不是真正的并行处理，只不过“有时候”切换速度快到人们感觉像是同时处理罢了。

线程适用的场合是非计算密集的场合，因为与其等待某个阻断作业完成，不如趁着等待的时间来运行其他线程。

对于计算密集型的任务，使用线程不见得会提高处理效率，反而容易因为解释器必须切换线程而耗费不必要的时间成本，使得效率变差。

如果要停止线程，那么必须自行实现，让线程跑完应有的流程。不仅是停止线程必须自行根据条件来实现，线程的暂停、重启也都必须根据需求来实现。

如果线程之间需要共享的是可变动状态的数据，就有可能发生竞争条件。如果要避免这类情况的发生，就必须在资源被变更与取用的关键程序代码中实现锁定机制。由于线程无法获取锁定时会造成阻断，因此不正确地使用 `Lock` 有可能造成性能低下，另一问题就是死锁。

`threading.Condition` 正如其名称所提示的，某个线程在通过 `acquire()` 获取锁定之后，如果需要在特定条件符合之前等待，那么可以调用 `wait()` 方法，这会释放锁定，如果其他线程的运行促成特定条件成立，那么可以调用同一 `threading.Condition` 实例的 `notify()`，通知等待条件的一个线程可获取锁定（也许有其他线程也正在等待），如果等待线程获取锁定，就会从上次调用 `wait()` 方法处继续执行。

对于计算密集型的运算，若能在一个新的进程并行运行，在如今计算机普遍都有多个核心处理器的情况下，就有机会运行得更快一些。

`subprocess` 模块可以让我们在执行 Python 程序的过程中产生新的子进程。

如果想要以子进程来执行函数，类似 `threading` 模块的 API 接口，那么可以使用 `multiprocessing` 模块。

为了在子进程执行时让 python 解释器安全地导入 `main` 模块，`if __name__ == '__main__':` 的测试是必要的，如果没有这行语句，就会引发 `RuntimeError` 错误。

建议在使用 `multiprocessing` 模块时不要共享状态，然而有时进程之间难免需要进行沟通，这时可以使用 `multiprocessing.Queue`，它在线程与进程之间实现了安全、必要的锁定机制。

`multiprocessing` 模块也提供了一些不同于 `threading` 模块的 API，例如 `Pool`，这可以创建一个工作者池，利用 `Pool` 实例可以派送任务并获取 `multiprocessing.pool.AsyncResult` 实例，在任务完成后获取结果。

课后练习

实践题

1. 如果有一个线程池可以分配线程来执行指定的函数，执行完后该线程类必须能重复使用，该线程类如何设计呢？

第 14 章

高级主题

学习目标

- 运用描述器
- 实现装饰器
- 定义 meta 类
- 使用相对导入

```
o2.py
> c:\workspace\pdb_demo>python -m pdb filter_demo2.py
> c:\workspace\pdb_demo\filter_demo2.py(1)<module>()
return len_greater_than_one
-> def filter_it(predicate, lt):
    Just in: enterpillar, sphinx
    At 5: filter_it(len_greater_than(5), 1)
    大于 7: (Rdb) c.l(len_greater_than(1), lt)
    (Pdb)
1 -> def filter_it(predicate, lt):
2     result = []
3     for elem in lt:
4         if predicate(elem):
5             result.append(elem)
6     return result
7
8 def len_greater_than(num):
9     def filter_it(predicate, lt):
10         return len([elem for elem in lt if predicate(elem)]) > num
11     return filter_it
12
13 if __name__ == '__main__':
14     lt = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
15     f = len_greater_than(5)
16     f(filter_it(lambda x: x > 5, lt), lt)
17     print result
```

```
(Pdb)
1 -> def filter_it(predicate, lt):
2     result = []
3     for elem in lt:
4         if predicate(elem):
5             result.append(elem)
6     return result
7
8 def len_greater_than(num):
9     def filter_it(predicate, lt):
10         return len([elem for elem in lt if predicate(elem)]) > num
11     return filter_it
12
13 if __name__ == '__main__':
14     lt = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
15     f = len_greater_than(5)
16     f(filter_it(lambda x: x > 5, lt), lt)
17     print result
```



14.1 属性控制

在第 5 章中，我们介绍过@property 的使用，也讨论过属性命名空间，实际上通过实例存取属性还有许多细节可以决定对象该如何做出反应。

14.1.1 描述器

一个对象可以被称为描述器 (Descriptor)，它必须拥有__get__()方法，以及选择性的__set__()、__delete__()方法，这三个方法的描述如下：

```
def __get__(self, instance, owner)
def __set__(self, instance, value)
def __delete__(self, instance)
```

在 Python 中，描述器就是一种对象，用来描述属性的获取、设置、删除该如何处理。当描述器实际成为某个类的属性成员时，对于类属性或者其实例属性的获取、设置或删除，将会交由描述器来决定如何处理（除了内建属性，如__class__等属性之外）。例如：

attributes descriptor.py

```
class Descriptor:
    def __get__(self, instance, owner):
        print(self, instance, owner, end = '\n\n')

    def __set__(self, instance, value):
        print(self, instance, value, end = '\n\n')

    def __delete__(self, instance):
        print(self, instance, end = '\n\n')

class Some:
    x = Descriptor()

s = Some()
s.x
s.x = 10
del s.x

some.x
```

范例中 Descriptor 类实现了__get__()、__set__()与__delete__()三个方法，符合描述器的协议，当 Descriptor 被指定给 Some 类的 x 属性时，Some 实例 s 的属性取值、设置或删除分别相当于进行了以下的操作：

```
Some.__dict__['x'].__get__(s, Some)
Some.__dict__['x'].__set__(s, 10)
Some.__dict__['x'].__delete__(s)
```

Some.x 这个取值操作相当于：

```
Some.__dict__['x'].__get__(None, Some)
```


因此，上面这个范例的执行结果为：

```
<__main__.Descriptor object at 0x01CC5A90> <__main__.Some object at 0x01D6C2F0> <class
'__main__.Some'>

<__main__.Descriptor object at 0x01CC5A90> <__main__.Some object at 0x01D6C2F0> 10

<__main__.Descriptor object at 0x01CC5A90> <__main__.Some object at 0x01D6C2F0>

<__main__.Descriptor object at 0x01CC5A90> None <class ' __main__.Some'>
```

在前面谈及属性命名空间时曾说过 `__dict__` 的作用，稍后我们将会介绍 `__getattr__()` 的作用。在结合描述器一并整理且试图获取某个属性时，完整的搜索顺序应该是：

1. 在产生实例的类 `__dict__` 中寻找是否有相符的属性名称。如果找到的是个描述器实例（也就是具有 `__get__()` 方法），且具有 `__set__()` 或 `__delete__()` 方法，若为取值，则返回 `__get__()` 方法的值；若为设置值，则调用 `__set__()`（没有这个方法则抛出 `AttributeError` 例外）；若为删除属性，则调用 `__delete__()`（没有这个方法则抛出 `AttributeError` 例外）；若描述器只具有 `__get__()`，则先进行第 2 步。
2. 在实例的 `__dict__` 中寻找是否有相符的属性名称。
3. 在产生实例的类 `__dict__` 中寻找是否有相符的属性名称。若不是描述器，则直接返回属性值。若是个描述器（此时一定只具有 `__get__()` 方法），则返回 `__get__()` 的值。
4. 若实例定义了 `__getattr__()`，则看 `__getattr__()` 如何处理，若没有定义 `__getattr__()`，则抛出 `AttributeError` 例外。

以上的流程可以做个简单的验证：

```
>>> class Desc:
...     def __get__(self, instance, owner):
...         print('instance', instance, 'owner', owner)
...     def __set__(self, instance, value):
...         print('instance', instance, 'value', value)
...
>>> class X:
...     x = Desc()
...
>>> x = X()
>>> x.x
instance <__main__.X object at 0x016BB350> owner <class '__main__.X'>
>>> x.x = 10
instance <__main__.X object at 0x016BB350> value 10
>>> x.__dict__['x'] = 10
>>> x.x
instance <__main__.X object at 0x016BB350> owner <class '__main__.X'>
>>> x.__dict__['x']
10
>>> del x.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: __delete__
>>>
```

上面的示范中，只要通过 `X` 实例直接存取 `x` 属性，都会由描述器来处理，为了绕过描述器，故意使用 `x.__dict__['x'] = 10` 来设置值，然而 `x.x` 时还是被描述器处理了，之后的 `x.__dict__['x']` 证

明，x 实例确实有个属性被设置为 10 了。

描述器的最基本协议是具备 `__get__()` 方法，若还具有 `__set__()` 或 `__delete__()` 方法或两者兼具，可进一步称为数据描述器（Data descriptor）。上面描述的就是数据描述器的行为。

相对地，仅有 `__get__()` 方法的描述器可进一步称为非数据描述器（Non-data descriptor）。对于非数据描述器，如果实例上有对应的属性，描述器就不会有操作。例如：

```
>>> class Desc:
...     def __get__(self, instance, owner):
...         print('instance', instance, 'owner', owner)
...
>>> class X:
...     x = Desc()
...
>>> x = X()
>>> x.x
instance <__main__.X object at 0x01E01FD0> owner <class '__main__.X'>
>>> x.x = 10
>>> x.x
10
>>> del x.x
>>> x.x
instance <__main__.X object at 0x01E01FD0> owner <class '__main__.X'>
>>>
```

在上面的范例中，一旦 X 的实例被指定了 x 属性值，就看不到描述器有所动作了。简单来说，数据描述器可以拦截对实例的属性获取、设置与删除操作，非数据描述器用来拦截通过实例获取类属性时的行为。

回顾 5.2.4 小节的内容，实际上 `@property` 是用来把对实例的属性存取转为调用 `@property` 标注的函数，这是一种数据描述器的行为，我们可以模拟类似的功能，例如：

attributes prop_demo.py

```
def prop(getter, setter, deleter):
    class PropDesc:
        def __get__(self, instance, owner):
            return getter(instance)
        def __set__(self, instance, value):
            setter(instance, value)
        def __delete__(self, instance):
            deleter(instance)

    return PropDesc()  # ① 返回描述器

class Ball:
    def __init__(self, radius):
        if radius <= 0:
            raise ValueError('必须是正数')
        self.__radius = radius

    def get_radius(self):
        return self.__radius

    def set_radius(self, radius):
        self.__radius = radius

    def del_radius(self):
```

```

del self.__radius

radius = prop(get_radius, set_radius, del_radius)  ← ②传入取值、设值等方法

ball = Ball(10)
print(ball.radius) # 显示 10
ball.radius = 5
print(ball.radius) # 显示 5

```

这个范例的重点在于将 `get_radius`、`set_radius`、`del_radius` 传入 `prop()`②，它会返回一个描述器①，这个描述器被指定为 `Ball` 类的 `radius`，因此对实例的 `radius` 属性存取都会通过描述器来处理，也就是调用传入的 `get_radius`、`set_radius` 或 `del_radius` 方法来处理。

14.1.2 定义 `__slots__`

如果想控制可以指定给对象的属性名称，那么可以在定义类时指定 `__slots__`，这个属性是个字符串列表，列出可指定给对象的属性名称。例如想限制 `Some` 的实例只能有 `a`、`b` 两个属性，可以如下编写程序：

```

>>> class Some:
...     __slots__ = ['a', 'b']
...
>>> Some.__dict__.keys()
['a', '__module__', 'b', '__slots__', '__doc__']
>>> s = Some()
>>> s.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: a
>>> s.a = 10
>>> s.a
10
>>> s.b = 20
>>> s.b
20
>>> s.c = 30
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Some' object has no attribute 'c'
>>>

```

在上面的范例中，虽然在 `__slots__` 中列出的属性存在于类的 `__dict__` 中，但在指定属性给实例之前，不可以直接存取该属性，而且只有 `__slots__` 中列出的属性才可以被指定给实例。

如果类定义时指定了 `__slots__`，那么从类构建出来的实例就不会具有 `__dict__` 属性。例如：

```

>>> s = Some()
>>> s.__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Some' object has no attribute '__dict__'
>>>

```

实际上，可以在 `__slots__` 中包括 `'__dict__'` 名称，让实例拥有 `__dict__` 属性。这么一来，若指定的属性名称不在 `__slots__` 的列表中，就会被放到自行指定的 `__dict__` 列表中，此时若要列出实例的全部属性，也就要同时包括 `__dict__` 与 `__slots__` 中列出的属性。例如：


```
>>> class Some:
...     __slots__ = ['a', 'b', '__dict__']
...
>>> s = Some()
>>> s.__dict__
{}
>>> s.a = 10
>>> s.b = 20
>>> s.c = 30
>>> s.__dict__
{'c': 30}
>>> for attr in list(s.__dict__) + s.__slots__:
...     print(attr, getattr(s, attr))
...
c 30
a 10
b 20
__dict__ {'c': 30}
>>>
```

实际上，`__slots__` 中的属性 Python 会将之实现为描述器，描述器会具有 `__get__()` 方法，以及选择性的 `__set__()`、`__delete__()` 方法，也就可以如下操作：

```
>>> class Some:
...     __slots__ = ['a', 'b']
...
>>> Some.__dict__.keys()
['a', '__module__', 'b', '__slots__', '__doc__']
>>> s = Some()
>>> Some.__dict__['a'].__set__(s, 10)
>>> Some.__dict__['a'].__get__(s, Some)
10
>>>
```

所以 `__slots__` 属性最好被作为类属性来使用，尤其是在有继承关系的场合中，父类中定义的 `__slots__` 仅可以通过父类来存取，子类的 `__slots__` 仅可以通过子类来存取。

在寻找实例上可设置的属性时，基本上会对照父类与子类中的 `__slots__` 列表。然而由于只有定义 `__slots__` 的类，其实例才不会有 `__dict__` 属性，因此若父类中没有定义 `__slots__`，子类即使定义了 `__slots__`，以子类构建出来的实例仍然会具有 `__dict__` 属性。

```
>>> class P:
...     pass
...
>>> class C(P):
...     __slots__ = ['c']
...
>>> o = C()
>>> o.a = 10
>>> o.b = 10
>>> o.c = 10
>>> o.__dict__
{'a': 10, 'b': 10}
>>>
```

反之亦然，如果父类定义了 `__slots__`，而子类没有定义自己的 `__slots__`，那么子类构建出来的实例也会有 `__dict__`。例如：

```
>>> class P:
...     __slots__ = ['c']
...
>>>
```

```
>>> class C(P):
...     pass
...
>>> o = C()
>>> o.a = 10
>>> o.b = 10
>>> o.c = 10
>>> o.__dict__
{'a': 10, 'b': 10}
>>>
```

`__slots__` 中的属性是由描述器来实现的，对于一些属性很少的对象来说，使用 `__slots__` 有可能增加一些性能，因为 `__dict__` 是字典对象，如果对象创建后仅设置很少的属性，对于空间就是种浪费；如果使用 `__slots__` 的列表（list）实现属性的存取，那么可能对性能有所帮助。

14.1.3 `__getattribute__()`、`__getattr__()`、`__setattr__()`、`__delattr__()`

对象本身可以定义 `__getattribute__()`、`__getattr__()`、`__setattr__()`、`__delattr__()` 等方法，以决定存取属性的行为，这些方法的描述如下：

```
def __getattribute__(self, name)
def __getattr__(self, name)
def __setattr__(self, name, value)
def __delattr__(self, name)
```

`__getattribute__()` 最容易解释，一旦定义了这个方法，任何属性的寻找都会被拦截，即使是那些 `__xxx__` 的内建属性名称。

`__getattr__()` 是作为寻找属性的最后一个机会。如果同时定义了 `__getattribute__()`、`__getattr__()`，那么在寻找属性时的顺序是：

1. 若定义了 `__getattribute__()`，则返回 `__getattribute__()` 的值。
2. 在产生实例的类 `__dict__` 中寻找是否有相符的属性名称。若找到且实际是个数据描述器，则返回 `__get__()` 方法的值。若是非数据描述器，则进行第 3 步。
3. 在实例的 `__dict__` 中寻找是否有相符的属性名称，若有则返回。
4. 在产生实例的类 `__dict__` 中寻找是否有相符的属性名称。若不是描述器则直接返回属性值。若是描述器（此时一定是只具有 `__get__()` 方法），则返回 `__get__()` 的值。
5. 若实例定义了 `__getattr__()`，则看 `__getattr__()` 如何处理，若没有定义 `__getattr__()`，则抛出 `AttributeError` 例外。

简单来说，获取属性顺序的记忆原则为：实例的 `__getattribute__()`、数据描述器的 `__get__()`、实例的 `__dict__`、非数据描述器的 `__get__()`、实例的 `__getattr__()`。

`__setattr__()` 的作用在于拦截所有对实例的属性设置。如果对实例进行设置属性的操作，那么设置的顺序如下：

1. 若定义了 `__setattr__()` 则调用；若没有则进行下一步。
2. 在产生实例的类中，看看 `__dict__` 是否有相符合的属性名称。若找到且实际是个数据描述器，则调用描述器的 `__set__()` 方法（若没有 `__set__()` 方法则丢出 `AttributeError`）；若不是则进行下一步。

3. 在实例的 `__dict__` 上设置属性与值。

简单来说，设置属性顺序记忆的原则是：实例的 `__setattr__()`、数据描述器的 `__set__()`、实例的 `__dict__`。

`__delattr__()` 的作用在于拦截所有对实例的属性设置。如果对实例有个删除属性的操作，那么删除的顺序如下：

1. 若定义了 `__delattr__()` 则调用，若没有则进行下一步。

2. 在产生实例的类中，看看 `__dict__` 是否有相符合的属性名称。若找到且实际是个数据描述器，则调用描述器的 `__delete__()` 方法（若没有 `__delete__()` 方法则抛出 `AttributeError` 例外）；若不是数据描述器则进行下一步。

3. 在实例的 `__dict__` 上寻找有无相符合的属性名称，若有则删除，若没有则抛出 `AttributeError` 例外。

简单来说，删除属性顺序记忆的原则是：实例的 `__delattr__()`、数据描述器的 `__delete__()`、实例的 `__dict__`。

14.2 装饰器

到目前为止，读者应该看过不少标注了，如 `@property`、`@staticmethod`、`@classmethod`、`@total_ordering` 等，在现有程序代码的适当位置添加标注，就能改变程序代码的行为，这类标注实际上被称为装饰器（Decorator），这一节就说明一下如何自定义装饰器。

14.2.1 函数装饰器

装饰器本质上就是一个函数，可接受函数且返回函数。这里以实际的例子来说明，假设我们设计了一个点餐程序，目前主餐有炸鸡，价格为 49 元。

```
def friedchicken():
    return 49.0

print(friedchicken()) # 49.0
```

之后在程序中其他几个地方都调用了 `friedchicken()` 函数，若现在我们打算增加附餐，以便客户点主餐的同时可以搭配附餐，程序代码该怎么做呢？修改 `friedchicken()` 函数还是另外增加一个 `friedchickenside1()` 函数？如果主餐不只有炸鸡，还有汉堡、意大利面等各式主餐呢？无论是修改各个主餐的相关函数还是新增各种 `xxxxside1()` 函数，显然都很麻烦且没有弹性。

别忘了，Python 中函数是一级值，一个函数可以接受函数并返回函数，因此可以这么编写：

```
def sidedish1(meal):
    return lambda: meal() + 30

def friedchicken():
    return 49.0

friedchicken = sidedish1(friedchicken)
```



```
print(friedchicken()) # 显示 79.0
```

sidedish1()接受函数对象,其中使用 lambda 创建一个函数对象,该函数对象执行传入的函数获取主餐价格,再加上附餐价格,sidedish1()返回此函数对象给 friedchicken 引用,所以之后执行的 friedchicken()就会是主餐加附餐的价格。

这仅仅只是传递函数的一个应用。在 Python 中还可以使用以下的语句:

decorators burgers.py

```
def sidedish1(meal):
    return lambda: meal() + 30

@sidedish1
def friedchicken():
    return 49.0

print(friedchicken()) # 显示 79.0
```

@之后名称引用的就是个函数,@sidedish1 这样的标注方式,让@sidedish1 对 friedchicken()函数加以装饰,在不改变 friedchicken()的行为下增加了附餐的行为,这类函数被称为装饰器,如果有必要,那么也可以进一步堆栈装饰器。

decorators burgers2.py

```
def sidedish1(meal):
    return lambda: meal() + 30

def sidedish2(meal):
    return lambda: meal() + 40

@sidedish1
@sidedish2
def friedchicken():
    return 49.0

print(friedchicken()) # 显示 119.0
```

最后执行的函数顺序就是从堆栈最底层开始往上层调用,就像下面的结果:

```
def sidedish1(meal):
    return lambda: meal() + 30

def sidedish2(meal):
    return lambda: meal() + 40

def friedchicken():
    return 49.0

friedchicken = sidedish1(sidedish2(friedchicken))

print(friedchicken())
```

如果装饰器语句需要带有参数,用来作为装饰器的函数,必须先以指定的参数执行一次,返回一个函数对象再来装饰指定的函数。例如下面这个带参数的装饰器。

```
@deco('param')
```

```
def func():
    pass
```

实际上执行时相当于:

```
func = deco('param')(func)
```

因此若要让点餐程序更有弹性一些, 可以这么设计:

decorators burgers3.py

```
def sidedish(number):
    return {
        1 : lambda meal: (lambda: meal() + 30),
        2 : lambda meal: (lambda: meal() + 40),
        3 : lambda meal: (lambda: meal() + 50),
        4 : lambda meal: (lambda: meal() + 60)
    }.get(number, lambda meal: (lambda: meal()))
```

```
@sidedish(2)
```

```
@sidedish(3)
```

```
def friedchicken():
    return 49.0
```

```
print(friedchicken()) # 显示 139.0
```

以上是使用 `lamdba` 创建函数, 设置为字典 (dict) 的值, 指定的号码会被当成键来获取对应要返回的函数, 若不易理解, 以下版本是较清楚的等效代码:

```
def sidedish(number):
    def dish1(meal):
        return lambda: meal() + 30

    def dish2(meal):
        return lambda: meal() + 40

    def dish3(meal):
        return lambda: meal() + 50

    def dish4(meal):
        return lambda: meal() + 60

    def nodish(meal):
        return lambda: meal()

    return {
        1 : dish1,
        2 : dish2,
        3 : dish3,
        4 : dish4
    }.get(number, nodish)
```

```
@sidedish(2)
```

```
@sidedish(3)
```

```
def friedchicken():
    return 49.0
```

```
print(friedchicken())
```

14.2.2 类装饰器

除了对函数进行装饰之外，也可以对类进行装饰，也就是类装饰器。首先编写以下的程序代码：

```
@decorator
class Some:
    pass
```

程序代码执行的效果相当于：

```
class Some:
    pass
Some = decorator(Some)
```

就如同函数装饰器实际上是一个接受函数并返回函数的函数，类装饰器是一个接受类并返回类的函数，因此先前的 friedchicken() 函数，因为设计上的考虑，打算定义为 FriedChicken，若想要对 FriedChicken 类进行装饰，使其加上附餐功能，则可以设计一个函数如下：

decorators burgers4.py

```
def sidedish1(cls):  ← ❶ 接受类
    class SideDishOne:
        def __init__(self):
            self.meal = cls()

        def getContent(self):
            return self.meal.getContent() + " | 可乐 | 薯条"

        def price(self):
            return self.meal.price() + 30.0

    return SideDishOne  ← ❷ 返回类

@sidedish1  ← ❸ 类装饰器
class FriedChicken:
    def getContent(self):
        return "不黑心炸鸡"

    def price(self):
        return 49.0

friedchicken = FriedChicken()
print(friedchicken.getContent())  # 不黑心炸鸡 | 可乐 | 薯条
print(friedchicken.price())      # 79.0
```

这个范例中的 sidedish1() 函数接受类❶，内部定义了一个 SideDishOne 类并返回❷，如果使用 @sidedish1 装饰 FriedChicken 类❸，那么实际上在构建类实例时会使用 sidedish1() 函数返回的类，因此最后调用 getContent() 或 price() 方法时都是 SideDishOne 定义的行为。

除了使用函数来定义装饰器之外，也可以使用类来定义装饰器，在了解如何使用类定义装饰器之前，要先了解 __call__() 方法的作用。

```
>>> class Some:
...     def __call__(self, *args):
```



```

...     for arg in args:
...         print(arg, end=' ')
...     print()
...
>>> s = Some()
>>> s(1)
1
>>> s(1, 2)
1 2
>>> s(1, 2, 3)
1 2 3
>>>

```

简单来说, 如果类中定义了 `__call__()` 方法, 那么构建的实例可以使用圆括号 `()` 来传入自变量, 此时会调用实例的 `__call__()` 方法, 就如上面的例子所示范的, 因此若要使用类来定义函数装饰器, 其做法为:

```

class decorator:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args):
        result = self.func()
        # 对 result 进行装饰 (返回)

@decorator
def some(arg):
    pass

some(1)

```

执行以上的程序片段, 其实相当于:

```

some = decorator(some)
some(1) # 调用 some.__call__(1)

```

也就是使用 `decorator` 创建实例, 之后对 `some` 实例的调用都会转换为对 `__call__()` 方法的调用, 因此若要以定义类的方式来对函数进行装饰, 可以如下编写:

decorators burgers5.py

```

class sidedish1:
    def __init__(self, func):
        self.func = func

    def __call__(self):
        return self.func() + 30

@sidedish1
def friedchicken():
    return 49.0

print(friedchicken()) # 79.0

```

若要以定义类的方式实现对类的装饰器, 则原理是:

```

class decorator:
    def __init__(self, clz):
        self.clz = clz

```

```

def __call__(self, *args):
    class wrapper:
        def __init__(self, *args):
            self.wrapped = self.clz(*args)
        # 其他的方法
    return wrapper(*args)

@decorator
class Some:
    pass

s = Some(1, 2)

```

执行以上的程序代码，效果相当于：

```

Some = decorator(Some)
s = Some(1, 2) # 其实是调用 Some.__call__(1, 2) 返回 wrapper 实例

```

例如，若使用定义类的方式对 FriedChicken 进行装饰，可以如下实现：

decorators burgers6.py

```

class sidedish1:
    def __init__(self, clz):
        self.clz = clz

    def __call__(self):
        class SideDishOne:
            def __init__(self, meal):
                self.meal = meal

            def getContent(self):
                return self.meal.getContent() + " | 可乐 | 薯条"

            def price(self):
                return self.meal.price() + 30.0

        return SideDishOne(self.clz())

@sidedish1
class FriedChicken:
    def getContent(self):
        return "不黑心炸鸡"

    def price(self):
        return 49.0

friedchicken = FriedChicken()
print(friedchicken.getContent()) # 不黑心炸鸡 | 可乐 | 薯条
print(friedchicken.price())     # 79.0

```

想要对一个函数或类进行装饰，可以选择的实现方式有函数与类两种，该选择哪种要根据需求而定。如果在实现装饰器时不需要有相关的状态与操作，那么就选择函数来实现；如果实现装饰器时需要考虑状态与操作之间的关系，那么就选择类来实现。

14.2.3 方法装饰器

除了直接对函数或类进行装饰之外，也可以对类中定义的方法进行装饰。可以使用函数或者类来实现，重点在于方法的第一个参数总是类的实例本身。

例如，若要以函数来实现方法装饰器，一个简单的例子如下：

```
def log(mth):
    def wrapper(self, a, b):
        print(self, a, b)
        return mth(self, a, b)
    return wrapper

class Some:
    @log
    def doIt(self, a, b):
        return a + b

s = Some()
print(s.doIt(1, 2))
```

粗体字的部分其实相当于在类中定义了：

```
doIt = log(doIt)
```

`wrapper` 的第一个 `self` 参数在上例中不可以省略，因为要用来接受实例。可以将上例设计得更通用一些，让 `@log` 装饰的对象不限于可接受两个自变量的方法。例如：

```
def log(mth):
    def wrapper(self, *args, **kwargs):
        print(self, args, kwargs)
        return mth(self, *args, **kwargs)
    return wrapper

class Some:
    @log
    def doIt(self, a, b):
        return a + b

s = Some()
print(s.doIt(1, 2))
```

以上是使用函数来实现方法装饰器的做法，接下来看看如何以类来实现方法装饰器。

如果我们想实现 `@staticmethod` 与 `@classmethod` 的功能，需要搭配描述器来实现。例如下面这个范例，实现 `@staticmth` 来模拟 `@staticmethod` 的功能。

```
decorators staticmth_demo.py
```

```
class staticmth: # 定义一个描述器
    def __init__(self, mth):
        self.mth = mth
```

```
    def __get__(self, instance, owner):
        return self.mth
```

```
class Some:
```



```

@staticmethod    # 相当于 doIt = staticmeth(doIt)
def doIt(a, b):
    print(a, b)

Some.doIt(1, 2) # 相当于 Some.__dict__['doIt'].__get__(None, Some)(1, 2)
s = Some()

# 以下相当于 Some.__dict__['doIt'].__get__(s, Some)(1)
# 所以以下会有错 TypeError: doIt() takes exactly 2 positional arguments ..
s.doIt(1)

```

`staticmethod` 类的 `__get__()` 只返回原来 `Some.doIt` 引用的函数对象，而不会作为实例绑定方法，因此就算通过实例调用了被 `@staticmethod` 装饰的方法，方法的第一个参数也不会被绑定为实例。

如果来实现 `@classmethod` 来模拟 `@classmethod` 的功能，则可以如下：

decorators classmeth_demo.py

```

class classmeth: # 定义一个描述器
    def __init__(self, mth):
        self.mth = mth

    def __get__(self, instance, owner):
        def wrapper(*arg, **kwargs):
            return self.mth(owner, *arg, **kwargs)
        return wrapper

class Other:
    @classmeth # 相当于 doIt = classmeth(doIt)
    def doIt(cls, a, b):
        print(cls, a, b)

Other.doIt(1, 2) # 相当于 Other.__dict__['doIt'].__get__(None, Other)(1, 2)
o = Other()
o.doIt(1, 2) # 相当于 Other.__dict__['doIt'].__get__(o, Other)(1, 2)

```

由于描述器协议中 `__get__()` 的第三个参数总是接受类实例，因此可以指定给原类中的方法作为第一个参数，从而实现 `@classmethod` 的功能。

以上只是以实现 `@staticmethod` 与 `@classmethod` 的功能作为示范，总之，以方法装饰器的原理为出发点，结合各种对象协议，就可以实现更多不同的功能。

14.3 Meta 类

在 6.1.4 小节中曾经看过如何定义抽象类，当时在定义类时还指定了 `metaclass` 为 `abc` 模块的 `ABCMeta` 类，我们可以自行定义 `meta` 类，用于决定类本身的创建、初始化以及其实例的创建与初始化，这一节将来看看它是怎么办到的。

14.3.1 认识 `type` 类

在 Python 中，可以使用类来定义对象的蓝图，每个对象实例本身都有个 `__class__` 属性，引用

至实例构建时使用的类，而类本身也有个 `__class__` 属性，那么它会引用到什么呢？

```
>>> class Some:
...     pass
...
>>> s = Some()
>>> s.__class__
<class '__main__.Some'>
>>> Some.__class__
<class 'type'>
>>>
```

可以看到，类的 `__class__` 引用到 `type` 类，每个类也是一个对象，是 `type` 类的实例。

在先前谈类装饰器时，我们曾经看过类中可以定义 `__call__()` 方法，当一个对象本身直接使用圆括号 `()` 来调用时，就会调用 `__call__()` 方法，既然类是个对象，构建对象实例时是在类名称之后接上圆括号，那么试着在类中调用 `__call__()` 会如何呢？

```
>>> class Some:
...     def __init__(self):
...         print('__init__')
...
>>> Some()
__init__
<__main__.Some object at 0x01D0BF10>
>>> Some.__call__()
__init__
<__main__.Some object at 0x01D16350>
>>>
```

通过 `Some.__call__()` 竟然调用了 `Some` 定义的 `__init__()`！这其实是 `type` 类定义的行为，每个类是 `type` 类的实例，这样想就完全符合 `__call__()` 方法的行为了。

既然每个类都是 `type` 类的实例，那么有没有办法编写程序代码直接使用 `type` 类来构建出类呢？可以！首先必须知道的是，使用 `type` 类构建类时必须指定三个自变量，分别是类名称（字符串）、类的父类（元组，`tuple`）与类的属性（字典，`dict`）。

使用 `type` 类创建的实例会拥有 `__call__()` 方法，假设 `type` 创建的实例为 `Some`，如果以 `Some(10)` 进行调用，等同于调用 `Some.__call__(10)`，这会调用 `Some` 的 `__init__()` 方法（如果定义了 `__new__()` 方法，那么会在 `__init__()` 之前调用）。

事实上，如果如下定义类：

```
class Some:
    s = 10
    def __init__(self, arg):
        self.arg = arg

    def doSome(self):
        self.arg += 1
```

Python 会在解析完类之后创建 `s` 名称引用至 10，创建 `__init__` 与 `doSome` 名称分别引用至各自定义的函数，然后调用 `type()` 来创建 `type` 类的实例并赋值给 `Some` 名称，也就是类似于：

```
Some = type('Some', (object,), {'__init__': __init__, 'doSome': doSome})
```

在 Python 中对象是类的实例，而类是 `type` 的实例，如果有方法能介入 `type` 创建实例与初始化的过程，就会有办法改变类的行为，这就是 `meta` 类的基本概念。

注意>>>

这与装饰器的概念不同，要对类使用装饰器时，类本身已经产生，也就是已经产生了 `type` 实例，然后才去装饰类的行为；`meta` 类直接介入 `type` 构建与初始化类的过程，时机点并不相同。

14.3.2 指定 metaclass

`type` 本身既然是个类，那么可以继承它吗？可以的，而且 `type` 类的子类一样可以指定类名称（字符串）、类的父类（元组，`tuple`）与类的属性（字典，`dict`）三个自变量，例如：

`metaclass_demo type_demo.py`

```
class SomeMeta(type):    # 继承 type 类
    def __new__(metaclass, clzname, parents, attrs):
        clz = super(metaclass, metaclass).__new__(metaclass, clzname, parents, attrs)
        print('SomeMeta __new__', metaclass, clzname, parents, attrs)
        return clz

    def __init__(clz, clzname, parents, attrs):
        super().__init__(clzname, parents, attrs)
        print('SomeMeta __init__', clz, clzname, parents, attrs)

Some = SomeMeta('Some', (object,), {'doSome': (lambda self, x: print(x))})

s = Some()
s.doSome(10)
```

在上面的例子中，继承 `type` 创建了 `SomeMeta`，并定义了 `__new__()` 与 `__init__()` 方法，`__new__()` 方法返回的实例才是 `Some` 最后会引用的类，接着 `__init__()` 进行该类的初始化，执行的结果如下：

```
SomeMeta __new__ <class '___main__.SomeMeta'> Some (<class 'object'>,) {'doSome': <function
<lambda> at 0x014B02B8>}
SomeMeta __init__ <class '___main__.Some'> Some (<class 'object'>,) {'doSome': <function
<lambda> at 0x014B02B8>}
10
```

上例中直接使用 `SomeMeta` 来构建类实例。实际上，可以在使用 `class` 定义类时指定 `metaclass` 为 `SomeMeta`。

```
>>> class Other(metaclass = type_demo.SomeMeta):
...     def doOther(self, x):
...         print(x)
...
SomeMeta __new__ <class 'type_demo.SomeMeta'> Other () {'__module__': '___main__',
'__qualname__': 'Other', 'doOther': <function Other.doOther at 0x01D31420>}
SomeMeta __init__ <class '___main__.Other'> Other () {'__module__': '___main__', '__qualname__':
'Other', 'doOther': <function Other.doOther at 0x01D31420>}
>>> other = Other()
>>> other.doOther(10)
10
>>>
```

一个继承了 `type` 的类可以作为 `meta` 类，`metaclass` 是个协议，若指定了 `metaclass` 的类，Python 在解析完类定义后会使用指定的 `metaclass` 来进行类的构建与初始化，其作用就像先前的范例。

如果使用 `class` 定义类时继承了某个父类，也想要指定 `metaclass`，可以如下编写：

```
class Other(Parent, metaclass = OtherMeta):
    pass
```

由于 `type` 本身也是一个类，使用类创建对象时：

```
x = X(arg)
```

实际上相当于：

```
x = X.__call__(arg)
```

`__call__()` 方法默认会调用 `X` 的 `__new__()` 与 `__init__()` 方法，若想改变一个类创建实例与初始化的流程，则可以在定义 `meta` 类时定义 `__call__()` 方法。

`metaclass_demo call_demo.py`

```
class SomeMeta(type):
    def __call__(cls, *args, **kwargs):
        print('call __new__')
        instance = cls.__new__(cls, *args, **kwargs)
        print('call __init__')
        cls.__init__(instance, *args, **kwargs)
        return instance
```

```
class Some(metaclass = SomeMeta):
```

```
    def __new__(cls):
        print('Some __new__')
        return object.__new__(cls)
```

```
    def __init__(self):
        print('Some __init__')
```

```
s = Some()
```

通过观察这个范例的执行结果，有助于了解一个类被调用、创建实例与初始化的过程。

```
call __new__
Some __new__
call __init__
Some __init__
```

基本上，`meta` 类就是 `type` 的子类，借助 `metaclass = MetaClass` 的协议可在类定义解析完后绕送至指定的 `meta` 类，可以定义 `meta` 类的 `__new__()` 方法，决定类如何创建，定义 `meta` 类的 `__init__()` 则可以决定类如何初始化，而定义 `meta` 类的 `__call__()` 方法，决定使用类来构建对象时，该如何进行对象的创建与初始化。

一个有趣的事实是，`metaclass` 并不仅仅可指定类，因为 Python 会调用指定对象的 `__call__()` 方法，并传入对象本身、类名称、父类信息与特性。因此，一个函数在调用时也可以使用函数对象的 `__call__()` 方法。

```
>>> def foo(arg):
...     print(arg)
...
>>> foo(10)
10
```

```
>>> foo.__call__(10)
10
>>>
```

知道了这点之后，应该就可以推断，metaclass 也可以指定函数：

metaclass_demo call_demo2.py

```
def metafunc(clzname, parents, attrs):
    print(clzname, parents, attrs)
    return type(clzname, parents, attrs)
```

```
class Some(metaclass = metafunc):
    def doSome(self):
        print('XD')
```

在上例中，函数的返回值将作为类，metafunc 的作用相当于 meta 类的 `__new__()` 与 `__init__()`，以此为出发点，metaclass 指定的对象可以是类、函数或任何对象，只要它具有 `__call__()` 方法。

提示 >>>

或许 Python 的 metaclass 这个名称应该叫 metaobject 比较正确！

14.3.3 `__abstractmethods__`

在 6.1.4 小节谈过抽象方法的定义，现在应该已经很清楚了，abc 模块的 ABCMeta 类就是刚才谈过的 meta 类，而 `@abstractmethod` 就是 14.2 小节介绍过的装饰器。既然如此，这里不妨来实现类似的功能。

在了解如何实现之前，必须先知道可以定义类的 `__abstractmethods__` 指明某些特性是抽象方法。例如：

```
>>> class AbstractX:
...     def doSome(self):
...         pass
...     def doOther(self):
...         pass
...
>>> AbstractX.__abstractmethods__ = frozenset({'doSome', 'doOther'})
>>> x = AbstractX()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class AbstractX with abstract methods doOther, doSome
>>>
```

在类创建之后可指定 `__abstractmethods__` 属性，`__abstractmethods__` 接受集合对象，集合对象中的字符串表明哪些方法是抽象方法，如果一个类的 `__abstractmethods__` 集合对象不为空，那么它就是个抽象类，不可以直接实例化。

子类不会看到父类的 `__abstractmethods__`。例如：

```
>>> class ConcreteX(AbstractX):
...     pass
...
>>> x = ConcreteX()
>>> ConcreteX.__abstractmethods__
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
AttributeError: __abstractmethods__
>>>
```

子类若没有实现父类所有的抽象方法，然而也想要定义抽象方法，则必须定义自己的 `__abstractmethods__`。

了解这些之后，就可以尝试模仿 `ABCMeta` 及 `abstractmethod()` 函数：

metaclass_demo myabc.py

```
def abstract(func):
    func.__isabstract__ = True # 表示这个函数是个抽象方法
    return func

class Abstract(type):
    def __new__(metaclass, clzname, parents, attrs):
        clz = super(metaclass, metaclass).__new__(metaclass, clzname, parents, attrs)

        # 类中定义的抽象方法
        abstracts = {name for name, value in attrs.items()
                     if getattr(value, "__isabstract__", False)}

        # 从父类中继承下来的抽象方法
        for parent in parents:
            for name in getattr(parent, "__abstractmethods__", set()):
                value = getattr(clz, name, None)
                if getattr(value, "__isabstract__", False):
                    abstracts.add(name)

        # 指定给 __abstractmethods__
        clz.__abstractmethods__ = frozenset(abstracts)

    return clz

class AbstractX(metaclass = Abstract):
    @abstract
    def doSome(self):
        pass

# TypeError: Can't instantiate abstract class AbstractX with abstract methods doSome
x = AbstractX()
```

在这个范例中，被自定义的 `@abstract` 标注的方法都会被设置 `__isabstract__` 属性。而在 `Abstract` 的 `__new__()` 中，会将具有 `__isabstract__` 属性的方法收集起来，并指定给类的 `__abstractmethods__`。

因此，定义类时 `metaclass` 指定了 `Abstract`，而且使用 `@abstract` 标注了抽象方法，`Abstract` 就不能直接用来构建实例了，子类也必须重新定义抽象方法，才能用来构建实例。

14.4 相对导入

在这一路学习 Python 的过程中，或许读者创建了许多模块，也许也开始应用软件包来管理这些模块了，我们在 2.2 小节讨论过模块与软件包管理，应该足以应付绝大多数需求。

不过，也许读者写的模块越来越多，模块可能会导入同一软件包中的其他模块内，而软件包

中也开始有子软件包。读者也许早就发现，单纯使用 2.2 小节中讨论过的模块与软件包管理，虽然可以应付需求，但是有些麻烦，不但模块名称必须避开标准链接库中的模块名称，若要导入其他软件包中的模块，总是要编写“又臭又长”的导入程序代码。

实际上，到目前为止使用的导入方式都是绝对导入 (**Abstract import**)，也就是指定软件包与模块的完整名称。Python 实际上还支持相对导入 (**Relative import**)。举个例子来说，如果有个软件包结构如下：

```
pkg1/
  __init__.py
  abc.py
  mno.py
  xyz.py
```

如果想在 xyz.py 中导入 abc 模块，在 xyz.py 中不能写 `import abc`，在 Python 3 中这会是绝对导入，也就是导入的会是标准链接库的 abc 模块，而不是 pkg1 中的 abc 模块。

如果想在 xyz.py 中导入 mno 模块，在 xyz.py 中不能写 `import mno`，这会引发 `ImportError` 错误，指出没有 mno 这个模块，如果要使用绝对导入，必须编写 `import pkg1.mno`，若要使用相对导入，则可以编写 `from . import mno`。

注意

在 Python 2.7 中，仍然可以 `import mno`，如果是在软件包中，就会导入同一软件包中的指定模块，这称为隐式相对导入 (**Implicit relative import**)。在 Python 3 中已经不能这么做了，若要导入同一软件包中的模块，必须使用 `from . import mno`，这称为显式相对导入 (**Explicit relative import**)，Python 3 删除了隐式相对导入的理由很容易理解，毕竟 Python 的哲学是“Explicit is better than implicit”。

如果想使用 mno 模块中的 `foo()` 函数，使用相对导入还可以编写 `from .mno import foo` 这样的方式，这么一来，就可以直接使用 `foo()` 来调用了。

相对导入的使用还可以让软件包下的模块在使用时更为方便。例如在某个程序中，若只是 `import pkg1`，只会执行 `__init__.py` 的内容，没办法直接使用 `pkg1.abc` 模块或其他模块；若想要使用 `pkg1.abc` 模块，必须再进行一次 `import pkg1.abc` 才行；如果想要在 `import pkg1` 时就能直接使用 `pkg1.abc` 模块或软件包中的其他模块，可以在 `pkg1` 的 `__init__.py` 中编写：

```
from . import abc
from . import mno
from . import xyz
```

这么一来，只要 `import pkg1`，就可以直接通过 `pkg1.abc`、`pkg2.mno`、`pkg1.xyz` 来使用模块，同样的手法也可以应用在软件包中还有子软件包的情况。例如，有个软件包与模块层级如下：

```
pkg1/
  __init__.py
  abc.py
  mno.py
  xyz.py
  sub_pkg/
    __init__.py
```

```
foo.py
orz.py
```

如果想要 import pkg1 之后可以直接使用 pkg1.abc、pkg2.mno、pkg1.xyz 模块，而且还能直接使用 pkg1.sub_pkg.foo、pkg1.sub_pkg.orz 模块，那么在 pkg1 的 __init__.py 中可以如下编写：

```
from . import abc
from . import mno
from . import xyz
from . import sub_pkg
```

在 pkg1.sub_pkg 的 __init__.py 中可以如下编写：

```
from . import foo
from . import orz
```

有一点小小的麻烦是，相对导入只能用在软件包之中，如果试图使用 python 解释器执行的某个模块中含有相对导入，就会引发 SystemError 错误。例如，若 xyz.py 中编写了：

```
from . import mno

def do_demo():
    mno.demo()

if __name__ == '__main__':
    do_demo()
```

直接使用 python xyz.py 执行时将会有以下错误：

```
Traceback (most recent call last):
  File "xyz.py", line 1, in <module>
    from . import mno
SystemError: Parent module '' not loaded, cannot perform relative import
```

如果模块与软件包管理日趋复杂，善用相对导入可以在管理上更为方便。如果读者重视这个主题，应该表明读者至少看完了本书，或者在 Python 的认知上也有一定的深度了。基于篇幅的限制，Python 中还有许多有趣的主题并不在本书中讲解，如果有机会，读者可以去找寻自己感兴趣的主体并加以探索。

14.5 重点复习

一个对象可以被描述为描述器 (Descriptor)，它必须拥有 __get__() 方法，以及选择性的 __set__()、__delete__() 方法。

数据描述器可以拦截对实例的属性获取、设置与删除行为，非数据描述器可以用来拦截通过实例获取类属性时的行为。

若想控制指定给对象的属性名称，则可以在定义类时指定 __slots__，这个属性是一个字符串列表，列出可指定给对象的属性名称。__slots__ 属性最好被作为类属性来使用。

一旦定义了 __getattr__() 方法，任何属性的寻找都会被拦截，即使是那些 __xxx__ 的内建属性名称。__getattr__() 是作为寻找属性的最后一个机会。

获取属性顺序记忆的原则为：实例的 __getattr__()、数据描述器的 __get__()、实例的

`__dict__`、非数据描述器的 `__get__()`、实例的 `__getattr__()`。

设置属性顺序记忆的原则是：实例的 `__setattr__()`、数据描述器的 `__set__()`、实例的 `__dict__`。

`__delattr__()` 的作用在于拦截所有对实例的属性设置。删除属性顺序记忆的原则是：实例的 `__delattr__()`、数据描述器的 `__delete__()`、实例的 `__dict__`。

装饰器本质上就是一个函数，可接受函数且返回函数。如果装饰器语句需要带有参数用来作为装饰器的函数，必须先以指定的参数执行一次，返回一个函数对象再来装饰指定的函数。

除了对函数进行装饰之外，也可以对类进行装饰，也就是类装饰器。

除了使用函数来定义装饰器之外，也可以使用类来定义装饰器。

除了直接对函数或类进行装饰之外，也可以对类中定义的方法进行装饰。可以选择使用函数或者类来实现，重点在于方法的第一个参数总是类的实例本身。

类的 `__class__` 引用到 `type` 类，每个类也是一个对象，是 `type` 类的实例。对象是类的实例，而类是 `type` 的实例，如果有方法能介入 `type` 创建实例与初始化的过程，就会有办法改变类的行为，这就是 `meta` 类的基本概念。

`meta` 类就是 `type` 的子类，通过 `metaclass = MetaClass` 的协议可在类定义解析完后绕送至指定的 `meta` 类，可以定义 `meta` 类的 `__new__()` 方法，决定类如何创建；定义 `meta` 类的 `__init__()` 则可以决定类如何初始化，而定义 `meta` 类的 `__call__()` 方法，决定使用类来构建对象时该如何进行对象的创建与初始化。

`metaclass` 指定的对象可以是类、函数或任何对象，只要它具有 `__call__()` 方法。

如果模块与软件包管理日趋复杂，善用相对导入可以在管理上更为方便。

课后练习

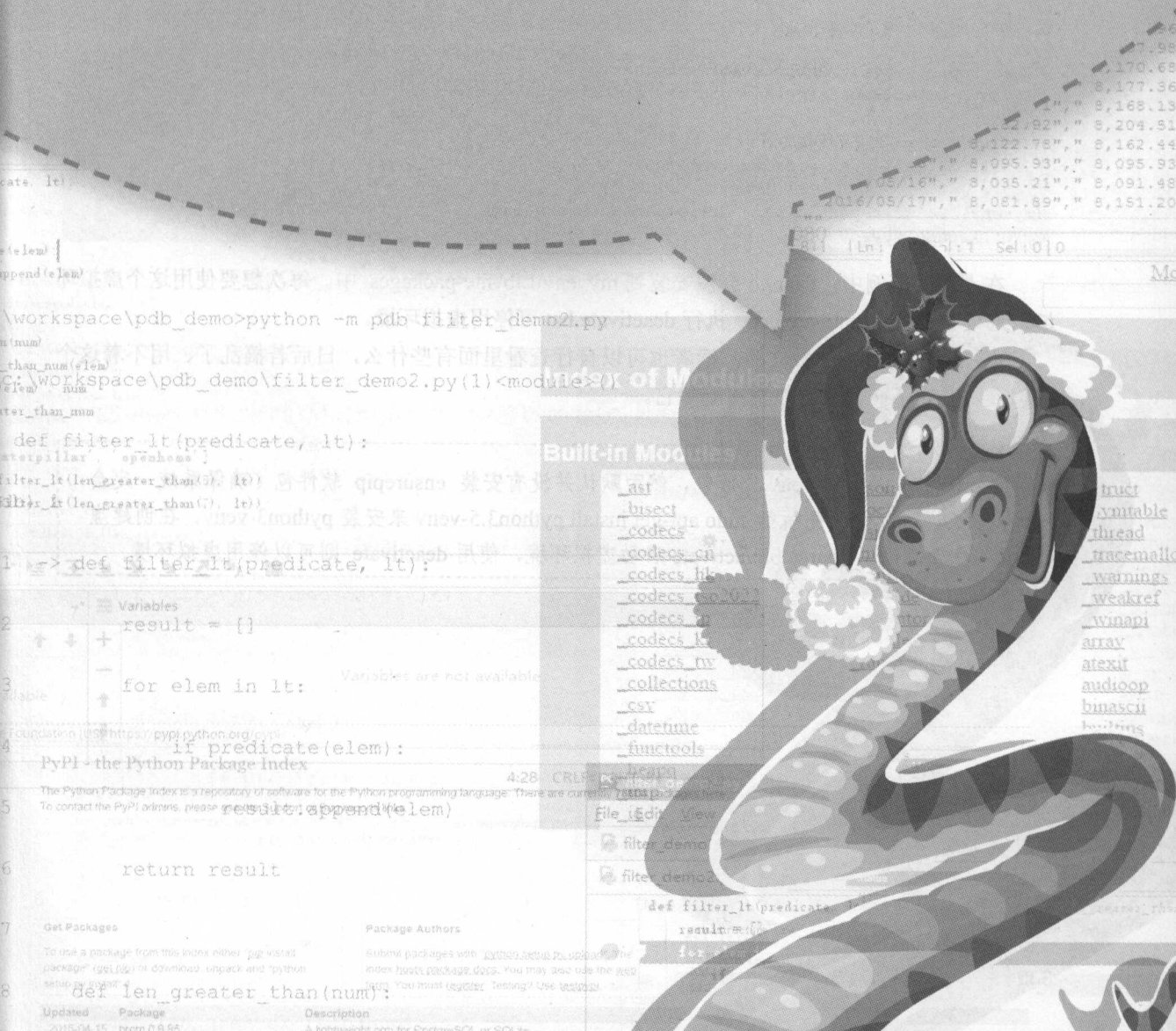
实践题

1. 在 6.2.2 小节曾经看过 `functools.total_ordering` 的使用，请实现一个 `total_ordering()` 函数作为装饰器，实现 `functools.total_ordering` 的功能。

附录 A venv

学习目标

- 创建虚拟环境



创建虚拟环境

Python 软件包不少都会直接安装到默认的系统路径中，例如安装 Django 时，在 Windows 中 pip 会将相关文件安装到 Python 目录的 Lib\site-packages\django 中。在初学 Python 的练习过程中，也许会安装许多软件包，如果不想都安装到系统路径中，或者不具备系统管理者权限而无法安装到系统路径，又或者需要在多个软件包版本之间切换，就会希望有个虚拟环境可以使用。

过去通常建议安装 Virtualenv，用以创建虚拟的 Python 环境，虚拟环境彼此之间互不干扰，也可避免搞乱 Python 主要的安装路径，从 Python 3.4 版本开始，内建了 venv 模块，可用来创建虚拟环境。

如果是在 Windows 下安装 Python 3.5，那么可以直接使用 venv。可以使用 `python -m venv myenv` 来创建虚拟环境，其中 myenv 是自行指定的名称，这个环境以目录为单位，因此会产生一个 myenv 目录，进入该目录后可以用 Scripts\activate.bat 启用虚拟环境，然后在其中用 pip 执行一些安装操作，使用 Scripts\deactivate.bat 可以停用虚拟环境。

```
C:\workspace>python -m venv myenv

C:\workspace>cd myenv

C:\workspace\myenv>Scripts\activate.bat
(myenv) C:\workspace\myenv>pip install django
Collecting django
  Using cached Django-1.9.7-py2.py3-none-any.whl
Installing collected packages: django

(myenv) C:\workspace\myenv>Scripts\deactivate.bat
C:\workspace\myenv>
```

在上面的范例中，Django 会被安装到 myenv\Lib\site-packages 中。每次想要使用这个虚拟环境，都要记得执行 activate.bat，执行 deactivate.bat 可停用虚拟环境。

虚拟环境的目录是独立的，读者也可以自行查看里面有什么，日后若搞乱了、用不着这个环境或看着它不爽，就可以直接删除它。

提示

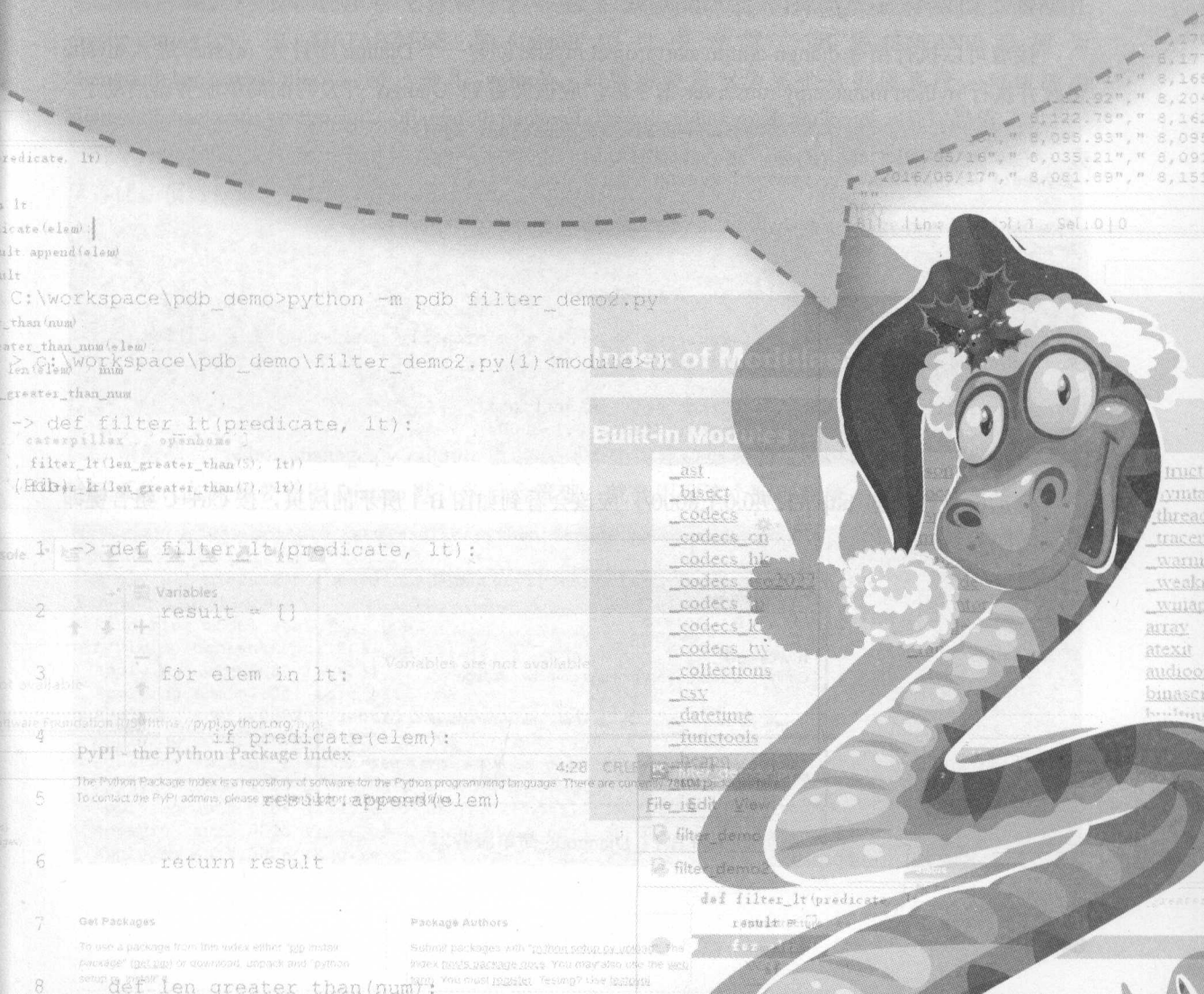
Ubuntu 中虽然也有 venv 软件包，然而默认并没有安装 ensurepip 软件包（确保系统一定会有 pip），我们必须先执行 `sudo apt-get install python3.5-venv` 来安装 python3-venv，在创建虚拟环境后，再用 `source bin/activate` 启动虚拟环境，使用 `deactivate` 则可以停用虚拟环境。

附录 B

Django 简介

学习目标

- 创建 Django 项目
- 创建 App 与模型
- 使用 ORM
- 创建简易窗体



B.1 Django 框架入门

Web 应用程序的编写是 Python 的一种应用, 对于入门 Python 也是个不错的练习对象, 因此在这个附录中将简要介绍如何使用 Python 来编写 Web 应用程序, 使用的框架(Framework)是 Django。

B.1.1 Django 起步走

这里使用 `venv` 创建一个虚拟环境, 并通过 `pip` 来安装 Django, 指令如下:

```
C:\workspace>python -m venv myenv
C:\workspace>cd myenv
C:\workspace\myenv>Scripts\activate.bat
(myenv) C:\workspace\myenv>pip install django
Collecting django
  Using cached Django-1.9.7-py2.py3-none-any.whl
Installing collected packages: django
(myenv) C:\workspace\myenv>
```

接着可以执行指令 `django-admin startproject mysite` 创建一个 Django 项目为 `mysite`, 进入 `mysite` 目录并执行 `python manage.py runserver` 指令后, 应该会看到 Django 开发时的简单服务器启动了。

```
(myenv) C:\workspace\myenv>django-admin startproject mysite
(myenv) C:\workspace\myenv>cd mysite
(myenv) C:\workspace\myenv\mysite>python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.
Justine 20, 2016 - 09:01:21
Django version 1.9.7, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

启动浏览器浏览 `http://127.0.0.1:8000/`, 应该会看到如图 B-1 所示的网页, 按 `Ctrl+C` 组合键即可关闭服务器。

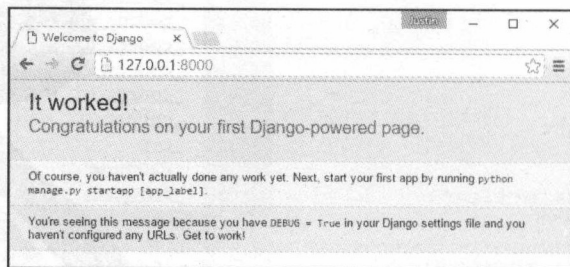


图 B-1 Django 的简单服务器

B.1.2 创建 App 与模型

至此，第一个 Django 项目已经创建成功了，那么项目中有哪些东西呢？我们现在来看看。

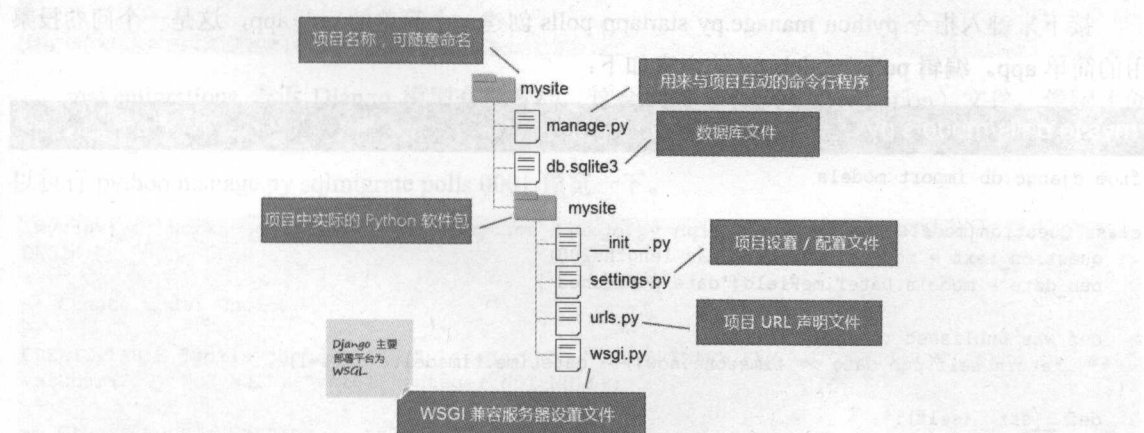


图 B-2 Django 项目结构

如图 B-2 所示，在项目文件夹中可以看到 `db.sqlite3` 数据库文件，存放的位置可在 `mysite/settings.py` 中 `DATABASES` 的 `'default'` 项目中设置，其中 `'ENGINE'` 设置为 `'django.db.backends.sqlite3'`，表示使用 `sqlite3`。如果想要设置为其他数据库系统，可以改为 `'django.db.backends.postgresql'`、`'django.db.backends.mysql'`、`'django.db.backends.oracle'` 等值。

`'NAME'` 可用来设置想要的数据库文件位置，默认值是 `os.path.join(BASE_DIR, 'db.sqlite3')`，表示存储在项目目录之下。

```
...
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
...
```

接着执行 `python manage.py migrate` 就会看到一些构建数据表的过程，然后会创建一个默认的经验证系统，如果想要使用 Django 默认的后台管理，就会用到这个验证系统。

```
(myvenv) C:\workspace\myvenv\mysite>python manage.py migrate
Operations to perform:
  Apply all migrations: sessions, contenttypes, admin, auth
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
```

```
Applying auth.0007_alter_validators_add_error_messages... OK
Applying sessions.0001_initial... OK
```

```
(myvenv) C:\workspace\myvenv\mysite>
```

接下来键入指令 `python manage.py startapp polls` 创建一个简单的 poll app，这是一个问卷投票用的简单 app。编辑 `polls/models.py` 的内容如下：

mysite polls/models.py

```
from django.db import models

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)

    def __str__(self):
        return self.question_text

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)

    def __str__(self):
        return self.choice_text
```

这创建了两个数据模型 `Question` 与 `Choice`，`Question` 中有 `question_text` 与 `pub_date` 两个字段，代表问题描述与发布日期，`was_published_recently()` 方法用来判断这个问题是不是最近一日内新发布的，`__str__()` 用来返回 `Question` 实例的字符串说明。`Choice` 用来记录投票选项，`question` 关联至问题（`Question` 实例），`choice_text` 是该问题的选项文字，`votes` 是投票数。

这个 app 刚创建，必须让当前项目知道有这个 app 的存在，这要在 `mysite/settings.py` 中设置，找到其中的 `'INSTALLED_APPS'`，在最后加入 `'polls'`。

mysite mysite/settings.py

```
...
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'polls'
]
...
```

接着可以执行 `python manage.py makemigrations polls`，会看到以下的信息：

```
(myvenv) C:\workspace\myvenv\mysite> python manage.py makemigrations polls
```



```

Migrations for 'polls':
  0001_initial.py:
    - Create model Choice
    - Create model Question
    - Add field question to choice

(myvenv) C:\workspace\myvenv\mysite>

```

`makemigrations` 告诉 Django 模型有所变动，这会创建一个迁移（migration）文件，像刚才创建的 `0001_initial.py`，当中说明了如何对数据库作出变更，如果想知道接下来会执行哪些 SQL，可以执行 `python manage.py sqlmigrate polls 0001` 预览一下。

```

(myvenv) C:\workspace\myvenv\mysite>python manage.py sqlmigrate polls 0001
BEGIN;
--
-- Create model Choice
--
CREATE TABLE "polls_choice" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "choice_text"
varchar(200) NOT NULL, "votes" integer NOT NULL);
--
-- Create model Question
--
CREATE TABLE "polls_question" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "question_text"
varchar(200) NOT NULL, "pub_date" datetime NOT NULL);
--
-- Add field question to choice
--
ALTER TABLE "polls_choice" RENAME TO "polls_choice_old";
CREATE TABLE "polls_choice" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "choice_text"
varchar(200) NOT NULL, "votes" integer NOT NULL, "question_id" integer NOT NULL REFERENCES
"polls_question" ("id"));
INSERT INTO "polls_choice" ("id", "votes", "question_id", "choice_text") SELECT "id", "votes",
NULL, "choice_text" FROM "polls_choice_old";
DROP TABLE "polls_choice_old";
CREATE INDEX "polls_choice_7aa0f6ee" ON "polls_choice" ("question_id");

COMMIT;

(myvenv) C:\workspace\myvenv\mysite>

```

如果没有问题，就执行 `python manage.py migrate` 完成迁移。

```

(myvenv) C:\workspace\myvenv\mysite>python manage.py migrate
Operations to perform:
  Apply all migrations: sessions, auth, contenttypes, polls, admin
Running migrations:
  Rendering model states... DONE
  Applying polls.0001_initial... OK

(myvenv) C:\workspace\myvenv\mysite>

```

接下来就可以实际操作刚才创建的 Question 与 Choice 了，相关的操作都会自动与数据库互动，例如作出变更或进行查询。

B.1.3 ORM 操作

刚才已经编写了模型程序代码，并且使用 Django 的迁移功能自动创建了数据库中的相关数据表，接下来我们就来操作模型与数据库吧！

查询全部数据

可以键入 `python manage.py shell` 指令，这会设置 `DJANGO_SETTINGS_MODULE` 环境变量，以便引用 Django 的 Python 模块，然后进入 Python 互动环境，在当中体验一些 API 的使用。

```
(myvenv) C:\workspace\myvenv\mysite>python manage.py shell
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from polls.models import Question, Choice
>>> Question.objects.all()
[]
>>> Choice.objects.all()
[]
>>>
```

一开始从 `polls.models` 模块中导入了 `Question` 与 `Choice` 类，若想查询全部的“问题”或“选项”，可以使用 `Question.objects.all()` 与 `Choice.objects.all()`，也就是在类名称之后接着 `objects.all()`。

可以看到，我们不用编写任何 SQL，Django 会自动转换对应的 SQL，获取结果然后包装为对象，不用自行进行对象与关系数据库之间的对应，这样的技术称之为 ORM（Object-Relational Mapping，对象关系映射），对于应用程序快速开发时非常便利。当然，当前“问题”与“选项”都没有任何信息，因此返回空列表（list）。

数据保存、字段查询与更新

接着在数据库进行数据的保存，直接来看看如何存储“问题”。

```
>>> from django.utils import timezone
>>> q = Question(question_text="What's new?", pub_date=timezone.now())
>>> q.save()
>>> q.id
1
>>> q.question_text
"What's new?"
>>> q.pub_date
datetime.datetime(2016, 6, 20, 1, 30, 49, 572671, tzinfo=<UTC>)
>>> q.question_text = "What's up?"
>>> q.save()
>>> Question.objects.all()
[<Question: What's up?>]
>>>
```

因为 `Question` 在保存时必须提供时间信息，因此从 `django.utils` 模块中导入了 `timezone`。要保存一个“问题”，只要创建 `Question` 实例，并调用 `save()` 就可以了，至于查询或更新相关的字段，也只是对属性进行操作。

特定条件查询

那么，如果有多个“问题”，想要进行条件过滤呢？可以使用 `filter()` 函数，例如：

```
>>> Question.objects.filter(id=1)
[<Question: What's up?>]
>>> Question.objects.filter(question_text__startswith='What')
[<Question: What's up?>]
>>> Question.objects.filter(id=2)
[]
>>> Question.objects.filter(question_text__startswith='What')
```

```
[<Question: What's up?>]
>>>
```

可以看到, `filter()` 函数会以列表 (list) 返回符合条件的数据, 如果只想取回一笔数据, 也可以使用 `get()`。不过, 若查询的条件不存在, `get()` 会引发 `DoesNotExist` 错误, 例如:

```
>>> Question.objects.get(pub_date__year=timezone.now().year)
<Question: What's up?>
>>> Question.objects.get(id=1)
<Question: What's up?>
>>> Question.objects.get(id=2)
Traceback (most recent call last):
 略...
polls.models.DoesNotExist: Question matching query does not exist.
>>> Question.objects.get(pk=1)
<Question: What's up?>
>>>
```

在查询主键时使用 `get(id=1)` 或 `get(pk=1)` 都可以。

创建关联与删除数据

一个“问题”会有多个“选项”, 来看看怎么创建两者之间的关联。

```
>>> q = Question.objects.get(pk=1)
>>> q.choice_set.all()
[]
>>> q.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> q.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = q.choice_set.create(choice_text='Just hacking again', votes=0)
>>> c.question
<Question: What's up?>
>>> q.choice_set.all()
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]
>>> q.choice_set.count()
3
>>> Choice.objects.filter(question__pub_date__year=timezone.now().year)
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]
>>> c = q.choice_set.filter(choice_text__startswith='Just hacking')
>>> c.delete()
(1, {'polls.Choice': 1})
>>>
```

这些操作都不需要太多解释, 若想进一步了解这类操作, 可以参考“Database API reference”¹。

B.1.4 编写第一个 View

了解使用 Django 基本的 ORM 操作之后, 现在来编写第一个 View, 打开 `polls/views.py`, 在当中编写如下的程序代码:

¹ Database API reference: docs.djangoproject.com/en/1.9/topics/db/queries/

mysite2 polls/views.py

```

from django.http import HttpResponse

def index(request):
    return HttpResponse('Hello, world. You\'re at the poll index.')

def detail(request, question_id):
    return HttpResponse(
        'You\'re looking at question {id}.'.format(id = question_id))

def results(request, question_id):
    return HttpResponse(
        'You\'re looking at the results of question {id}.'.format(id = question_id))

def vote(request, question_id):
    return HttpResponse(
        'You\'re voting on question {id}.'.format(id = question_id))

```

这里的 4 个函数将对对应到不同的 URL 请求，目前只是简单地显示些字符串。每个函数的第一个参数实际上是 `HttpRequest` 实例，封装了关于请求的相关数据，有些 URL 请求会带有 `question_id` 请求参数，这可以在函数的第二个参数获取请求值。

每个 URL 请求该如何对应到函数，可以在 `polls` 目录下创建一个 `urls.py` 文件进行定义。

mysite2 polls/urls.py

```

from django.conf.urls import url

from . import views

urlpatterns = [
    # ex: /polls/
    url(r'^$', views.index, name='index'),
    # ex: /polls/5/
    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
    # ex: /polls/5/results/
    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
    # ex: /polls/5/vote/
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]

```

`url` 的第一个参数接受正则表达式，用来定义 URL 模式，第二个参数表示应该对应到哪个函数，第三个参数用来定义这个 URL 模式的名称，某些地方若要引用这个定义，可以通过名称来指定引用。

如果仔细看上面的正则表达式定义，就会发现并没有定义 `polls` 前置名称，实际上这是在 `mysite` 目录中的 `urls.py` 定义。例如：

mysite2 mysite/urls.py

```

from django.conf.urls import url, include
from django.contrib import admin

urlpatterns = [
    url(r'^polls/', include('polls.urls')),
    url(r'^admin/', admin.site.urls),
]

```

这个 `urls.py` 定义了全名的 URL 对应，在上面可以看到定义了 `polls` 前置名称，接下来的规则包括在 `polls.urls` 中，也就是刚才在 `polls` 目录定义的 `urls.py` 中。

完成以上定义后，可以键入指令 `python manage.py runserver`，然后分别用浏览器请求访问不同的网站，我们应该会看到如图 B-3 所示的结果。

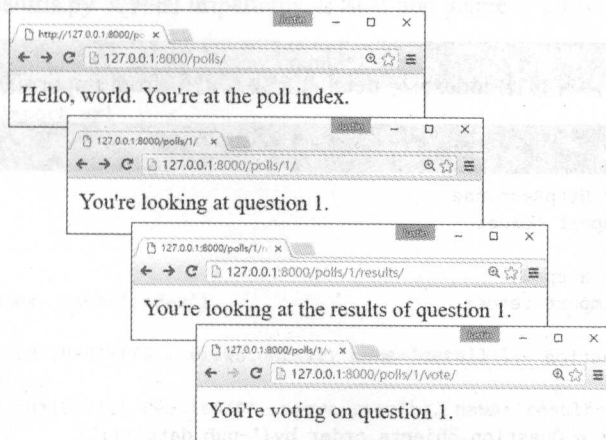


图 B-3 Django 中的第一组网页

B.1.5 使用模板系统

之前我们在 `polls/views.py` 中编写响应结果，不过实际的网页组织不建议编写在这当中。想想看，如果想要 HTML 输出，那么直接在 `polls/views.py` 中编写 HTML 输出，程序代码与 HTML 纠结在一起会是什么样的混乱结果。

建议 `views.py` 当中只使用 Python 程序代码来准备网页中动态的数据部分，但不包括网页组织以及相关的显示逻辑。可以使用 Django 的模板系统将网页组织以及相关的显示逻辑从 Python 程序代码中抽离出来。

在 `polls` 目录中创建一个 `templates` 目录，Django 会在这个目录中寻找模板，在 `templates` 目录中创建另一个名为 `polls` 的目录，并在其中创建一个名为 `index.html` 的文件。

也就是说，现在创建了一个模板文件 `polls/templates/polls/index.html`，接着将以下的程序代码放入模板之中。

`mysite3 polls/templates/polls/index.html`

```
{% if latest_question_list %}
<ul>
  {% for question in latest_question_list %}
    <li><a href="/polls/{{ question.id }}">{{ question.question_text }}</a></li>
  {% endfor %}
</ul>
{% else %}
  <p>No polls are available.</p>
{% endif %}
```

再创建一个名为 `detail.html` 的文件，并编写以下的程序代码：

mysite3 polls/templates/polls/detail.html

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
  <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

打开 polls/views.py, 并修改 index() 与 detail() 函数, 记得 from import 的部分也要一致。

mysite3 polls/views.py

```
from django.http import HttpResponseRedirect
from django.template import loader

from django.http import Http404
from django.shortcuts import render

from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    template = loader.get_template('polls/index.html')
    context = {
        'latest_question_list': latest_question_list,
    }
    return HttpResponseRedirect(template.render(context, request))

def detail(request, question_id):
    try:
        question = Question.objects.get(pk=question_id)
    except Question.DoesNotExist:
        raise Http404("Question does not exist")
    return render(request, 'polls/detail.html', {'question': question})

# 其他程序代码不变...
```

其中 'latest_question_list' 用来设置变量名称, render() 函数第二个自变量 'polls/index.html' 用来设置要显示的模板文件名。

执行 python manage.py runserver 之后, 应该可以在使用浏览器请求访问相关网址时看到如图 B-4 所示的页面。

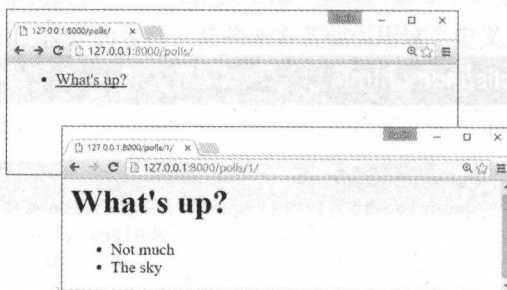


图 B-4 使用模型系统生成的网页

B.1.6 创建简易窗体

在当前的模板文件中，每个 URL 都有固定写好的路径。实际上，模板中可以使用 `url` 这个函数，通过命名空间等的指定来动态产生 URL。

首先，在当前 `polls/urls.py` 文件的 `urlpatterns` 前加上 `app_name = 'polls'`。

```
mysite4 polls/urls.py

from django.conf.urls import url

from . import views

app_name = 'polls'
urlpatterns = [
    # ex: /polls/
    url(r'^$', views.index, name='index'),
    # ex: /polls/5/
    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
    # ex: /polls/5/results/
    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
    # ex: /polls/5/vote/
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

接下来就可以在模板中使用 `url` 与命名空间的设置。例如修改 `polls/templates/polls/index.html` 模板。

```
mysite4 polls/templates/polls/index.html

{% if latest_question_list %}
<ul>
{% for question in latest_question_list %}
<li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a></li>
{% endfor %}
</ul>
{% else %}
<p>No polls are available.</p>
{% endif %}
```

接着要创建一个简易窗体，修改 `polls/templates/polls/detail.html`，包括 HTML 的 `<form>` 标签内容。

```
mysite4 polls/templates/polls/detail.html

<h1>{{ question.question_text }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
{% for choice in question.choice_set.all %}
<input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />
<label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```

在 `polls/views.py` 中增加以下内容并修改 `results()` 及 `vote()`，让 `results()` 可以根据请求的 `question_id` 与指定的模板文件绘制网页，而 `results()` 用来获取 `question_id` 更新选项的结果。

mysite4 polls/views.py

```
from django.template import loader
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect, HttpResponse
from django.core.urlresolvers import reverse

from .models import Choice, Question

# ... 其他程序代码

def results(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/results.html', {'question': question})

def vote(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    try:
        selected_choice = question.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        return render(request, 'polls/detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        return HttpResponseRedirect(reverse('polls:results', args=(question.id,)))
```

其中 `try..except..else` 部分，如果 `try` 区块中没有任何错误发生，就会执行 `else` 区块，而 `reverse('polls:results', args=(question.id,))` 会返回 `'polls/3/results'` 这样的字符串，也就是当选项设置完成后，直接重新定向到问题的投票结果网页。

当然，必须创建 `polls/templates/polls/results.html` 模板文件。

mysite4 polls/templates/polls/results.html

```
<h1>{{ question.question_text }}</h1>

<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</li>
{% endfor %}
</ul>

<a href="{% url 'polls:detail' question.id %}">Vote again?</a>
```

接着我们可以尝试连接网站并在上面投票，应该就可以看到如图 B-5 所示的结果。

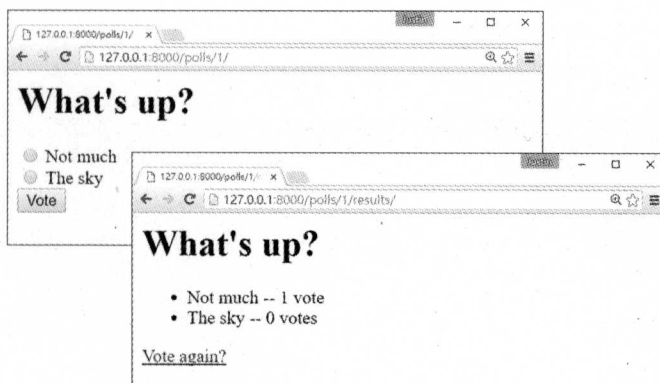
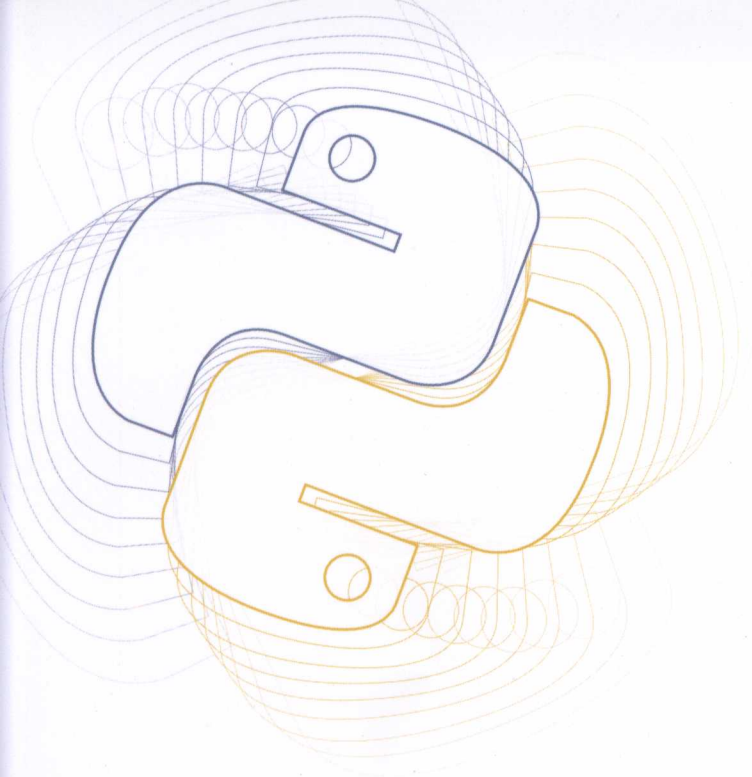


图 B-5 简易窗体



- 本书是作者在Python教学中针对学生学习时遇到的概念、应用实战等问题的经验总结。
- 基于Python 3.5编写，涵盖Python 3.0到3.5的实用特性。
- 从Python标准函数库的源码分析与探讨，了解标准函数库中各种语法的应用。
- 涵盖abc、collection.abc、datetime、pdb、unittest、timeit、threading、subprocess multiprocessing等标准函数库的实用模块。
- 对于描述器、装饰器、meta类的实践等进阶主题进行详细探讨，并以标准函数库中@staticmethod、@abstractmethod等功能的模仿作为实际应用的对象。
- 特以Lab图标注了重点范例，以便掌握学习重点。

清华大学出版社数字出版网站

WQBook  
www.wqbook.com

ISBN 978-7-302-45786-2



9 787302 457862 >

定价：59.00元